Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles

Fabien Hermenier

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ECOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATERIAUX

Année 2010

N° attribué par la bibliothèque

Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Informatique

Présentée
et soutenue publiquement par

Fabien Hermenier

Le 26 novembre 2009 à l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes

Devant le jury ci-dessous :

Président	:	José Martinez, Professeur	Université de Nantes
Rapporteurs	:	Bertil Folliot, Professeur	Université Paris VI
		Olivier Gruber, Professeur	Université de Grenoble
Examinateur	:	Christine Morin, Directeur de recherche	INRIA Rennes
Directeur de thèse	:	Gilles Muller, Directeur de recherche	INRIA Rocquencourt
Responsable Scientifique	:	Jean-Marc Menaud, Maître-assistant	École des Mines de Nantes

Équipe d'accueil : ASCOLA, INRIA, LINA UMR CNRS 6241
Laboratoire d'accueil : Département Informatique de l'École des Mines de Nantes
La Chantrerie – 4, rue Alfred Kastler – 44 307 Nantes

ED : .....

Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles

Online Management of Jobs in Clusters using Virtual Machines

Fabien Hermenier

Université de Nantes

Remerciements

À ceux qui m'ont supporté durant ces trois années,
À ceux qui m'ont « supporté » durant ces trois années.

Protocole oblige, d'abord la famille.

À mes parents, qui m'ont permis de suivre les études que j'ai souhaité entreprendre sans jamais remettre en cause mes choix. La route n'aura pas réellement été droite et, malgré un sens de l'orientation déficient, j'ai pu finalement atteindre un lieu qui me convient.

À ma famille, pour avoir supporté durant ces trois années, mes doutes, mes moments de mort cérébrale et ma quantité limitée de vacances.

À Sophie, pour sa patience quasi illimitée (j'ai pu dépasser la borne sup. à quelques reprises) et sa façon de supporter les questions que je me pose à moi-même lorsque je lui parle. Merci pour le temps accordé.

Ensuite la *Legen -wait for it- dary team* : Sophie, Xavier, Étienne, Rémi, Nicolas, Charles, Naren, Philippe, Julien, Aurélien, Guillaume (en espérant n'oublier personne).

Pour toutes les discussions insensées, mais péremptoires, que nous avons abordées,

Pour toutes les discussions sensées que nous avons pu sabordées.

Pour ne pas avoir fuit devant ma passion pour les choses ridicules et insignifiantes de la vie.

On continue avec l'environnement.

À la cafétéria, qui propose des Kinder surprise à la vente. Cela m'a permis d'avoir un bureau continuellement décoré avec la plus grande classe. Un bémol cependant avec la série des « animaux joueurs de foot » de l'été 2008 qui a dégradé sérieusement la qualité globale de ma collection.

Au secrétariat pour avoir fait preuve de compassion devant mon incapacité à envoyer un fax, ou à remplir correctement le moindre papier administratif.

Et on finit sur un brin de science.

Aux membres du jury, pour leurs commentaires et nombreuses questions qui m'ont permis d'ouvrir de nouvelles perspectives sur mon travail.

À Julia Lawall, pour sa rigueur et sa capacité hallucinante pour révéler des problèmes que j'aurais mis un temps fou à enfouir.

Enfin, à Jean-Marc Menaud et Gilles Muller, pour m'avoir encadré durant ces trois années et pour toutes les conversations, plus ou moins agitées, que nous avons partagées. Leurs commentaires et leur expertise m'ont permis de pousser toujours plus loin ma curiosité scientifique sans pour autant partir dans tous les sens.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

Le travail collaboratif a pour vocation d'associer plusieurs personnes à l'exécution d'une même tâche dans l'optique de réduire le temps nécessaire à son achèvement. Les personnes impliquées travaillent alors de concert sur ses différentes composantes, plus ou moins indépendantes. Parmi les différents critères de succès de cette parallélisation du travail, on compte la capacité de collaboration entre les participants, mais également l'intelligence dans la répartition du travail. Une mauvaise orchestration de ce partitionnement peut par exemple demander à une personne de réaliser un travail pour lequel elle n'est pas qualifiée. Cette affectation aura pour conséquence un travail de mauvaise qualité ou un temps de réalisation plus long que prévu pouvant affecter à terme l'achèvement de la tâche. Un bon partitionnement du travail implique de considérer à la fois les capacités, les qualités et la disponibilité des personnes afin d'obtenir le meilleur de chacun et d'achever la tâche au plus vite.

Le concept de travail collaboratif s'applique également en informatique dans le contexte des applications parallèles qui peuvent exécuter leurs différents processus sur plusieurs processeurs ou plusieurs machines. Ce modèle permet d'agréger une grande quantité de ressources et d'exécuter ainsi des opérations complexes comme des calculs scientifiques par exemple. Pour exécuter de telles applications, l'approche actuelle consiste à utiliser des grappes de serveurs, des architectures composées de plusieurs dizaines ou centaines de machines, appelées nœuds, interconnectées par un réseau performant. Chaque nœud dispose d'une quantité finie de ressources physiques (capacité CPU, mémoire, ...) et logicielles qu'il met à disposition des utilisateurs pour que ceux-ci exécutent leurs applications.

Lorsqu'un utilisateur souhaite exécuter une application parallèle sur une grappe, il s'assure d'abord que son application est compatible avec l'environnement de la grappe. En cas d'incompatibilité avec le matériel, le système d'exploitation ou certaines bibliothèques, il peut être nécessaire d'adapter l'application en recompilant son code source ou en le modifiant. Traditionnellement, les tâches ne sont pas exécutées directement par l'utilisateur mais par un service dédié sur la grappe, le gestionnaire de ressources. Lorsqu'un utilisateur souhaite exécuter son application, il soumet au gestionnaire de ressources une tâche, une description abstraite de l'application à exécuter. Le résultat de l'exécution par le gestionnaire des ressources est retourné par la suite. La description fournie par l'utilisateur renseigne, entre autres, le chemin vers l'application à exécuter, ainsi que les quantités de ressources requises pour exécuter chacun des processus composant l'application. Certaines applications ont des besoins en ressources constants et fixés dès le développement de l'application par l'utilisateur. Ce dernier spécifie alors directement les besoins en ressources de son application. D'autres applications offrent plus de souplesse et s'adaptent au moment de leur lancement à la quantité de ressources attribuée. Finalement, certaines applications ont des besoins en ressources variant dynamiquement au cours du temps. Dans ces situations, l'utilisateur précise simplement des quantités de ressources correspondant aux besoins maximums estimés de son application.

Le gestionnaire de ressources est un service chargé d'orchestrer l'exécution de toutes les tâches depuis le moment de leur soumission jusqu'au moment de leur terminaison. Il sélectionne en permanence parmi l'ensemble des tâches soumises, des tâches qu'il peut exécuter sur la grappe en s'assurant que pour celles-ci, une place est disponible sur les nœuds de la grappe pour chacun de leurs processus. Pour cela, le gestionnaire de ressources se base sur la description des besoins en ressources de la tâche et cherche parmi les nœuds, une partition de ressources pouvant accueillir chaque processus de la tâche. Une fois la partition

allouée à chacun des processus, le gestionnaire de ressources déploie l'application sur les différents nœuds et lance son exécution. Si la taille d'une partition allouée à un processus est inférieure à ses besoins, les performances globales de l'application seront dégradées ou son exécution sera compromise. À l'inverse, si la taille de la partition allouée est trop grande, le processus n'utilisera pas nécessairement la totalité de la mémoire ou des processeurs. Une certaine quantité de ressources sera donc inutilement réservée et potentiellement perdue pour l'exécution d'une autre tâche en attente de ressources pour son exécution.

Sélectionner les tâches à exécuter parmi l'ensemble des tâches soumises et trouver un nœud d'accueil pour chacun des processus, correspondent à des problèmes classiques respectivement d'ordonnancement en ligne de tâches et de placement. Pour résoudre ces problèmes, les gestionnaires de ressources proposent différents algorithmes qui cherchent à utiliser au mieux les ressources de la grappe en favorisant certains objectifs. Un algorithme d'ordonnancement va, par exemple, tenter d'exécuter les tâches le plus tôt possible pour que les utilisateurs attendent le moins longtemps avant de récupérer le résultat de leurs applications. Un algorithme de placement peut quant à lui chercher à exécuter les tâches sur le minimum de nœuds possible. Il est alors possible d'économiser de l'énergie en éteignant les nœuds inutilisés.

Une utilisation efficace des ressources de la grappe implique d'abord que les utilisateurs qui soumettent des tâches réalisent une description des besoins des ressources la plus proche possible des besoins réels, même en cas de besoins variables. Une utilisation optimale des ressources implique ensuite une stratégie d'ordonnancement et de placement des tâches la plus efficace possible, en utilisant au mieux les capacités de la grappe. Lorsque les ressources sont allouées à la volée, en fonction des besoins des tâches par exemple, il importe alors de pouvoir remettre en cause dynamiquement l'ordonnancement des tâches et leur placement pour empêcher à la fois une surcharge de la grappe, mais également une sous-utilisation de celle-ci. Le gestionnaire de ressources peut alors être amené à arrêter temporairement des tâches, à relancer de nouvelles tâches ou changer le placement courant des processus.

1.1 Objectifs

La gestion dynamique des tâches consiste à adapter à la volée les ressources attribuées aux processus des tâches en cours d'exécution, reconfigurer le placement des processus ou encore suspendre temporairement l'exécution d'une tâche, au profit d'une autre plus prioritaire par exemple. Les gestionnaires de ressources basés sur cette gestion dynamique des tâches peuvent optimiser en continu l'orchestration des tâches en allouant leurs ressources à la volée, en fonction de leurs besoins. Il existe une grande quantité de tels algorithmes d'ordonnancement et, bien que leurs concepts de base soient semblables, les administrateurs doivent tout de même adapter les algorithmes à leur grappe. Cela consiste à spécialiser leur fonctionnement en tenant compte à la fois des particularités des tâches soumises, mais également de différentes contraintes liées à l'ordonnancement des tâches ou à leur placement. Simultanément, le développement des primitives liées à la gestion dynamique des tâches est limité par la nature même des processus et le manque d'isolation de l'environnement utilisateur. Ainsi, dans la pratique, le développement et l'utilisation de stratégies à base de gestion dynamique des tâches dans les grappes est limité d'une part par l'inadaptivité de l'environnement et d'autre part, par la difficulté d'adapter les algorithmes d'ordonnancement et de placement.

Dans cette thèse, nous nous sommes fixés comme objectif de faciliter le développement de gestionnaires de ressources reposant sur une gestion dynamique des tâches. Pour mener à bien cet objectif, nous pensons qu'il est nécessaire de revoir l'architecture des grappes, notamment l'environnement d'exécution des tâches. Les environnement actuels, exécutant les tâches sur des systèmes d'exploitation standard, limitent la possibilité d'exécuter leurs applications dans leur environnement natif. Par ailleurs la manipulation des processus à la volée y est délicate. Nous pensons également qu'il est nécessaire de fournir un environnement flexible, où les administrateurs peuvent définir facilement des problèmes d'ordonnancement et de placement adaptés aux spécificités de leur grappe et des tâches à exécuter. Finalement, nous considérons que l'environnement doit être de fournir aux administrateurs la possibilité d'optimiser en continu l'ordonnancement et le placement des tâches dans la grappe.

1.2 Contributions

Ce document décrit notre apport dans le domaine de la gestion dynamique des tâches dans les grappes. Notre contribution porte principalement sur trois axes : la définition d'un environnement, Entropy, pour la gestion dynamique des tâches à base de machines virtuelles ; le développement de mécanismes assurant une reconfiguration dynamique des tâches dans les grappes, fiable et efficace ; l'étude pratique de deux scénarios répondant à des besoins concrets et l'implémentation de solutions au sein d'Entropy : la consolidation dynamique pour réduire les frais de fonctionnement des grappes et l'ordonnancement de tâches dans les grappes.

1.2.1 Entropy

Entropy est un système autonome et flexible pour la manipulation de machines virtuelles dans les grappes. Les machines virtuelles proposent aux utilisateurs des environnements semblables à une machine physique, complètement isolés les uns des autres, mais dont les ressources, virtuelles, sont contrôlées par un hyperviseur. Dans ce contexte, l'utilisateur embarque chaque processus de l'application à exécuter dans une machine virtuelle. Il dispose alors d'un contrôle total de son environnement logiciel et peut exécuter son application dans son environnement d'exécution originel. L'hyperviseur fournit ensuite les primitives nécessaires à la gestion dynamique des tâches en manipulant directement les machines virtuelles à la place des processus.

Entropy est développé suivant le principe de l'informatique autonome qui considère qu'une application doit pouvoir d'elle même s'adapter à son environnement. Dans notre contexte, Entropy réalise en continu une auto-adaptation de l'ordonnancement des tâches et de leur placement. Dans la pratique, après une observation de l'état courant de la grappe, un module de décision décide d'une nouvelle configuration des machines virtuelles améliorant l'utilisation des ressources. Un module de planification définit ensuite un plan décrivant la transition entre la configuration courante des machines virtuelles et la configuration calculé décidé par le module de décision. Ce plan est ensuite exécuté pour achever l'auto-optimisation.

La flexibilité d'Entropy provient de la possibilité donnée à l'administrateur de développer entre autre ses propres modules de décision afin d'exécuter ses propres algorithmes d'ordonnancement et de placement. Pour faciliter le développement de tels algorithmes, nous nous somme orientés vers une approche à base de programmation par contraintes pour assurer l'écriture de stratégies d'ordonnancement et de placement flexibles. Cette approche permet en effet de modéliser et de résoudre de tels problèmes combinatoires complexes, en les composants d'une conjonction de contraintes à satisfaire. En contre-partie, l'utilisation d'une telle approche d'optimisation exacte implique généralement un temps de calcul des solutions supérieur aux approches heuristiques ad-hoc usuelles. L'intérêt de l'approche est alors justifié quand si la qualité des résultats obtenus compense ce temps de calcul.

Entropy est un logiciel publié sous licence LGPL [LGP07] depuis janvier 2009. Son code source et ses binaires sont disponibles sur le site internet du projet [ENT].

1.2.2 Reconfiguration dynamique de machines virtuelles

Nous avons montré dans un premier temps l'intérêt d'utiliser la migration de machines virtuelles pour réduire la consommation énergétique de grappes [4]. L'approche a ensuite été généralisée en considérant d'autres primitives dédiées à la manipulation des machines virtuelles (lancement, arrêt, suspension et reprise). Pour passer d'une configuration de machines virtuelles à un autre, il est nécessaire d'exécuter plusieurs actions qui manipulent l'état et la position des machines virtuelles dans la grappe. L'exécution de certaines de ces actions est cependant soumise à des pré-conditions instaurant des dépendances entre les actions. Il est alors nécessaire de planifier les différentes actions composant une reconfiguration afin de s'assurer de la faisabilité du processus. Nous avons donc d'abord proposé un algorithme détectant ces dépendances et calculant un plan d'exécution assurant la faisabilité de la reconfiguration. De premières évaluations ont ensuite révélé que le temps d'exécution d'une reconfiguration était un facteur clef qu'il fallait considérer pour obtenir un système réactif. Nous avons alors proposé une approche globale pour la réduction du temps de ré-agencement de reconfiguration reposant encore une fois sur la programmation par contraintes.

1.2.3 Applications

Afin de valider notre approche pour la gestion dynamique des tâches, nous avons développé avec Entropy deux cas d'utilisation proposant des solutions à des problèmes actuels concrets.

Consolidation dynamique de machines virtuelles Nous avons dans un premier temps développé un module de décision concentrant en continue les machines virtuelles des tâches en cours d'exécution sur un nombre de nœuds minimum. Notre approche considère la capacité en ressources CPU et mémoire de chaque nœud et les besoins courants en ressources CPU et mémoire de chaque machine virtuelle pour calculer un placement où toute les machines virtuelles peuvent accéder à des quantités de ressources satisfaisantes, tout en étant hébergées sur un nombre minimum de nœuds. La reconfiguration du placement des machines virtuelles est alors réalisée par une série de migrations. Cette application a été implémentée dans un premier prototype dédié à la réduction de la consommation énergétique des grappes en éteignant les nœuds non-utilisés et en les rallumant uniquement en cas de besoin [4]. L'approche a ensuite été étendue lors du développement d'Entropy avec une approche à base de programmation par contraintes, utilisant le module de planification et d'optimisation pour réduire la durée d'exécution des migrations tout en assurant leur bonne exécution [1].

Ordonnancement de tâches Le principe de séparation des préoccupations dans Entropy ainsi que l'isolation du mécanisme de reconfiguration a mis en avant un support potentiel pour le développement d'algorithmes d'ordonnancement. Dans ce contexte, l'administrateur de la grappe développe un algorithme indiquant la liste des tâches à exécuter. Le module de planification d'Entropy assure alors de manière autonome la transition entre l'état courant des tâches de la grappe et l'état souhaité par l'administrateur. Cette approche tend à simplifier le développement d'algorithmes d'ordonnancement pour grappes en masquant à l'administrateur les problèmes liés à la réorganisation des tâches tout en lui faisant profiter des possibilités offertes par la gestion dynamique des tâches. Afin de valider ce concept, nous avons développé un algorithme d'ordonnancement exécutant les tâches au plus tôt [2, 5]. Cet algorithme améliore le taux d'utilisation des ressources de la grappe et tend à réduire le temps nécessaire à l'exécution des tâches.

1.3 Organisation du document

Ce document est structuré en deux parties. La première partie décrit le contexte de notre étude. Nous y présentons dans le chapitre 2 un état de l'art relatif à la gestion des ressources dans les grappes de serveurs en discutant des différents modèles de tâches et des différentes stratégies d'ordonnancement et de placement couramment employé. Cette analyse exhibe le besoin d'un environnement manipulant à la volée les tâches et leurs ressources en résolvant des problèmes combinatoires complexes. Dans le chapitre 3, nous présentons un état de l'art dédié aux machines virtuelles système en discutant d'abord des différentes approches proposées actuellement dans la littérature. Nous réalisons ensuite un parallèle entre les mécanismes manipulant les processus et les tâches dans les grappes et les mécanismes manipulant les machines virtuelles. Finalement, nous terminons cette première partie en présentant dans le chapitre 4 les principes de la programmation par contraintes, l'approche que nous avons choisi pour définir et résoudre de manière exacte et flexible les problèmes d'ordonnancement, de placement et la planification des actions de reconfiguration.

La seconde partie de ce document est dédiée à la présentation des différents éléments de contribution de cette thèse. Dans le chapitre 5 nous définissons notre problématique en exprimant d'abord les différents besoins d'un environnement dédié à la gestion dynamique des tâches. Nous décrivons ensuite notre proposition, un système autonome à base de machines virtuelles et dont les algorithmes d'ordonnancement repose sur une approche à base de programmation par contraintes. Nous présentons dans le chapitre 6 les différents composants de notre prototype Entropy. Dans le chapitre 7, nous détaillons un cas d'utilisation réalisant de la consolidation dynamique, une solution pour réduire les coûts de fonctionnement des grappes ou augmenter leur capacité d'accueil. Nous présentons dans le chapitre 8 les différents problèmes liés à la manipulation à la volée des machines virtuelles et notre approche pour assurer une reconfiguration sûre et efficace de celles-ci. Finalement, nous présentons dans le chapitre 9 un deuxième cas d'utilisation d'Entropy par le développement de stratégies d'ordonnancement de tâches.

Le chapitre 10 de ce document est dédié à l'évaluation expérimentale de nos différents éléments de contributions. Nous discutons d'abord au travers de micro-évaluations des capacités de résolution de différents modules d'Entropy. Cette évaluation montre alors que notre approche calcule des solutions dont la qualité égale ou dépasse la qualité d'approches heuristiques courantes mais nécessite en contrepartie un temps de calcul plus long. Nous discutons ensuite de différentes expérimentations réalisées sur des grappes de serveurs qui démontrent que l'amélioration du processus de reconfiguration des machines virtuelles compense efficacement le temps de calcul de nos solutions et permet une utilisation de notre approche pour réaliser des systèmes reposant sur une gestion dynamique des tâches. Finalement, nous concluons ce manuscrit en réalisant d'abord un bilan de cette thèse puis en discutant de différentes perspectives.

1.4 Diffusion scientifique

Les différents travaux présentés dans ce document ont fait l'objet de diverses publications listées ci-dessous.

Conférence internationale avec comité de lecture

[1] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy : a consolidation manager for clusters. In *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50, New York, NY, USA, 2009. ACM.

Conférence nationale avec comité de lecture

[2] Fabien Hermenier, Adrien Lèbre, and Jean-Marc Menaud. Changement de Contexte pour Tâches Virtualisées dans les Grappes. In *Proc. of 7ème Conférence Francophone sur les Systèmes d'Exploitation (CFSE07)*, Toulouse, France, sep. 2009.

[3] Fabien Hermenier, Xavier Lorca, Hadrien Cambazard, Jean-Marc Menaud, and Narendra Jussien. Reconfiguration automatique du placement dans les grilles de calculs dirigée par des objectifs. In *Proc. of 6ème Conférence Francophone sur les Systèmes d'Exploitation (CFSE06)*, Fribourg, Swiss, fev. 2008.

Atelier international avec comité de lecture

[4] Fabien Hermenier, Nicolas Loriant, and Jean-Marc Menaud. Power management in grid computing with Xen. In *Proceedings of 2006 on XEN in HPC Cluster and Grid Computing Environments (XHPC06)*, number 4331 in LNCS, pages 407–416, Sorento, Italy, 2006. Springer Verlag.

Rapport de recherche

[5] Fabien Hermenier, Adrien Lèbre, and Jean-Marc Menaud. Cluster-Wide Context Switch of Virtualized Jobs. Research Report RR-6929, INRIA, 2009

Poster

[6] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy : a consolidation manager for clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI) - Poster Session*, San Diego, December 2008.

Première partie

Contexte de l'étude

Chapitre 2

Les gestionnaires de ressources pour grappes

Où nous discutons des modèles de tâches et de partitions de ressources utilisable dans des grappes de serveurs. Ces approches autorisent différents niveaux de flexibilité quant à leur possibilité de s'adapter à la gestion dynamique des tâches. Nous discutons également des différentes stratégies d'ordonnancement ayant pour objectif d'exécuter les tâches, soumisent par les utilisateurs, selon des critères précis et permettant une utilisation plus ou moins efficace des ressources de la grappe.

Sommaire

DURANT les 30 dernières années, l'architecture des systèmes dédiés à l'exécution d'applications massivement parallèles, comme les applications scientifiques par exemple, a évolué. D'une architecture monolithique composée de supercalculateurs et conçue sur mesure, nous sommes passé à une architecture appelée *grappe de serveurs*, interconnectant différents ordinateurs utilisant du matériel standard par un réseau rapide. La première réalisation concrète d'une grappe apparaît en 1994 avec le projet Beowulf de la NASA[SBS+95]. La grappe était composée de 16 machines avec un processeur à 100 MHz reliées par un réseau Ethernet 10 Mbps. Cette grappe sera utilisée pour la simulation de phénomènes physiques et l'acquisition de données. Comme sur les supercalculateurs, les applications s'exécutant sur une grappe sont des applications parallèles, c'est à dire pouvant utiliser plusieurs processeurs simultanément. Cependant, alors qu'il est envisageable de faire communiquer ensemble les différents composants d'une application sur un supercalculateur, il est nécessaire de limiter au maximum cette communication sur les grappes. En effet, sur un supercalculateur, les messages qui circulent entre les différents composants d'une application utilisent des bus ayant une latence et un débit bien meilleurs que sur une grappe où les différents messages circulent sur un réseau.

Contrairement aux supercalculateurs, l'architecture des grappes est faite pour évoluer et il est possible d'augmenter simplement la capacité de traitement en ajoutant de nouvelles machines. Le coût de mise en place d'une grappe est également nettement inférieur à celui d'un supercalculateur. Par exemple, Barroso *et al.* [BDH03] décrivent en 2003 que le coût d'acquisition d'une grappe de 88 machines avec

chacune 2 processeurs Intel Xeon à 2 GHz, 2 Go mémoire vive et un disque dur de 80 Go est trois fois moins élevé qu'un serveur proposant 8 processeurs, 64 Go de mémoire vive et 8 To d'espace de stockage. Les grappes apparaissent alors comme étant moins coûteuses à l'achat, avec un rapport prix/performance supérieure tout en disposant d'une architecture plus évolutive. Au fil des années, ce modèle architectural est devenu le modèle de référence pour le calcul haute performances [TOP] ou l'hébergement massif d'applications parallèles.

Les grappes permettent généralement à plusieurs personnes ou institutions d'exécuter plusieurs applications simultanément. Chaque utilisateur souhaitant exécuter une application sur la grappe soumet alors une tâche. La soumission contient une description des ressources qu'il souhaite obtenir (quantité de mémoire vive, nombre de processeurs, librairies, ...) afin d'exécuter différentes applications. Afin d'assurer une bonne utilisation de la grappe, il convient de coordonner les différentes tâches à exécuter par l'intermédiaire d'un gestionnaire de ressources, un service qui planifie l'exécution des différentes tâches en veillant à satisfaire les besoins spécifiés par les utilisateurs. Un de leurs enjeux est de pouvoir occuper au mieux les ressources de la grappe afin d'exécuter les différentes tâches des utilisateurs aussi vite que possible.

Nous dressons dans ce chapitre un état de l'art relatif à la gestion des ressources dans les grappes de serveurs. Après une description de l'architecture générale d'une grappe et de ses différents composants physiques ou logiciels, nous décrivons les différents modèles de tâches et de partitionnement des ressources. Nous présentons enfin les différentes stratégies d'ordonnancement des tâches utilisant un partage spatial ou temporel des ressources.

2.1 Architecture d'une grappe

Une grappe de serveurs est un ensemble d'ordinateurs (appelés nœuds) regroupés dans un même lieu et interconnectés par un réseau rapide. Traditionnellement, l'ensemble des nœuds composant une grappe est segmenté et chaque sous-ensemble se voit attribué un certain rôle. Les nœuds de calcul sont les machines dédiées à l'exécution des tâches des utilisateurs. Sur ces machines, l'environnement d'exécution est réduit au minimum afin de limiter la quantité de services fonctionnant en arrière plan. Les nœuds de services exécutent les différents services dédiés à la gestion de la grappe. On y trouve le plus souvent un nœud « frontal » servant de point d'entrée aux utilisateurs souhaitant utiliser la grappe. D'autres nœuds servent à l'hébergement de services spécifiques comme le service de supervision des nœuds et d'authentification. Pour les grappes composées de nœuds ayant une architecture matérielle ou un environnement logiciel précis, des nœuds peuvent être dédiés à la compilation des applications des utilisateurs en vue d'une adaptation de celles-ci aux spécificité de l'environnement.

Les différents nœuds d'une grappe sont reliés par un réseau rapide. La majorité des grappes utilise un réseau de type Giga Ethernet, permettant d'obtenir des liens ayant un débit théorique de 1 Gbps [FJ99] tandis que la norme récente du 10-Gigabit Ethernet [FHN+03], encore peu déployée, permet d'obtenir des liens 10 Gbps. Le réseau Ethernet propose avant tout une solution bon marché pour l'interconnexion des nœuds, cette solution est cependant la moins performante. En effet, le protocole Ethernet et les couches supérieures de la pile réseau ne sont pas faites pour des réseaux haute performance : la couche protocolaire consomme une partie non-négligeable de la bande passante, le protocole d'émission de paquets n'est pas déterministe et il n'empêche pas les collisions. On observe alors une latence élevée et un débit réel sous charge inférieure aux débits théoriques.

Différentes infrastructures réseaux ont été développées et proposent des solutions dédiées aux environnements grappes telles que les réseaux à base d'équipements Myrinet [BCF+95], InfiniBand [Pfi01] ou Quatrics [PFH+02]. Ces architectures permettent d'obtenir des performances meilleures qu'avec un réseau Ethernet en proposant un protocole simplifié, une latence très faible, une bande passante très proche des performances théoriques et des pilotes permettant d'utiliser les interfaces réseaux en évitant la partie supérieure de la pile réseau (par exemple, les interfaces Infiniband permettent d'obtenir une bande passante allant jusqu'à 40 Gbps). Le prix de ces équipements étant nettement plus élevé que du matériel Ethernet, leur utilisation est souvent limitée aux grappes dédiées à l'hébergement d'applications haute performances où la latence du réseau joue un rôle prépondérant.

La gestion du stockage des données dans les grappes de serveurs dépend de l'utilisation qui est faite de celles-ci et du niveau de performance à atteindre. Dans la majorité des cas, les fichiers temporaires

des applications n'ont pas besoins d'être partagés, ceux-ci peuvent être stockées sur les disques locaux des nœuds de calcul afin d'obtenir un temps d'accès minimal. Dans les cas où les données doivent être partagées entre différentes machines, il est souvent conseillé de mettre à disposition différents serveurs de fichiers utilisant des nœuds dédiés et du matériel spécialisé. Afin d'améliorer les performances, les différents volumes locaux peuvent être agrégés pour d'augmenter la disponibilité des données et les performances en lecture et en écriture. Dans certains contextes d'utilisation, un réseau entier de serveurs peut être dédié au stockage des données, mettant à disposition un système de fichiers distribué combinant différents volumes disques distants.

Le gestionnaire de ressources fait parti des services assurant la gestion des tâches. Il est responsable des tâches s'exécutant sur la grappe. Une tâche décrit une application qu'un utilisateur souhaite exécuter sur la grappe. Une application peut utiliser plusieurs processus, potentiellement inter-communicants et nécessitant des ressources physiques (capacité de calcul par le biais de processeurs, mémoire vive, ...) ou logicielles (librairies, licences, ...). Le gestionnaire de ressources doit donc assurer l'exécution de l'ensemble des tâches alimenté en continu par les soumissions des utilisateurs. Il doit permettre un taux d'occupation des ressources le plus élevé possible, tout en satisfaisant les requêtes des utilisateurs et différentes politiques d'ordonnancement. Nous détaillerons les différents modèles de tâches en section 2.2 tandis que les principes d'ordonnancement des tâches seront discutés en section 2.3.

Bien que le gestionnaire de ressources dispose de sa propre vue de l'état de chaque ressource (réservée ou libre), il est souvent couplé à un service de supervision plus global comme Ganglia [MCC04], NWS [WSH99] ou Nagios [ID07]. Ces systèmes de supervision permettent de suivre automatiquement l'état de la grappe. Cela peut simplement consister à surveiller l'état des ressources dans le temps afin de repérer des pics d'activités, l'évolution de l'espace disque, des interfaces réseau, de la mémoire ou des processeurs. Il permet aussi de repérer des nœuds défaillants. Ce service peut également être chargé de superviser d'autres applications, comme le gestionnaire de ressources, afin de suivre les différentes tâches en cours d'exécution.

2.2 Modèles de tâches

Une tâche est composée d'un ensemble de processus, potentiellement inter-communicants et exécutant une ou plusieurs applications. Exécuter une tâche sur une grappe consiste alors à exécuter l'ensemble des processus la constituant sur une architecture distribuée disposant de plusieurs processeurs. En 1996, Feitelson *et al.* [FR96] proposent une classification de différents modèles de tâches en les caractérisant par le niveau de flexibilité des tâches et du sous-ensemble de ressources physiques alloué aux tâches appelé partition.

2.2.1 Flexibilité des tâches

Le niveau de flexibilité d'une tâche désigne sa capacité à utiliser les ressources qui lui sont allouées par le gestionnaire de ressources. On distingue quatre niveaux de flexibilité :

- **Les tâches rigides**
  Ces tâches utilisent une quantité constante de ressources durant toute leur exécution. La quantité de ressources est connue à l'avance par l'utilisateur. La tâche ne peut pas fonctionner avec une quantité de ressources inférieure et n'utilisera pas les ressources accordées en supplément ;

- **Les tâches malléables**
  Ces tâches s'adaptent à la quantité de ressources allouée au début de leur exécution. Cette quantité peut éventuellement être redéfinie explicitement durant l'exécution de la tâche ;

- **Les tâches évolutives**
  Ces tâches ont un profil d'exécution passant par différentes phases d'une durée variable. Lorsque la tâche change de phase, sa consommation en ressources est susceptible de varier ;

- **Les tâches préemptibles**
  Traditionnellement, une tâche est exécutée jusqu'à sa terminaison normale ou forcée, provoquée par le gestionnaire de ressources. Les tâches préemptibles au contraire peuvent être suspendues dans un état consistant puis être relancées ultérieurement. Le résultat de cette capture d'état peut être conservé simplement en mémoire vive, ou être stocké sur disque. Si la tâche a été suspendue sur disque alors elle ne nécessite plus de ressources et la quantité de ressources qui lui a été allouée peut être libérée. Si la tâche a été suspendue en mémoire alors elle requiert toujours une certaine quantité de mémoire. La préemption locale consiste à relancer la tâche sur la machine l'ayant suspendu tandis que la migration consiste à transférer la tâche sur une autre machine avant de le relancer. Le temps de migration de la tâche peut être court si les différents nœuds de la grappe partagent la mémoire au travers du réseau. Dans le cas contraire, le contenu de la mémoire doit être transféré sur la machine destination en plus de son état et de son environnement logiciel. Cette action peut nécessiter un temps significatif et impacter sur les performances des nœuds impliqués. Il importe alors de considérer ce coût dans le gestionnaire de ressources et de le minimiser.

2.2.2 Partitionnement des ressources

Le gestionnaire de ressources a pour objectif d'allouer des ressources aux tâches soumises afin de pouvoir les exécuter. Il est donc chargé de sélectionner dans l'ensemble des ressources disponibles sur la grappe (ressources processeurs et mémoire principalement) une partition satisfaisant les besoins de la tâche. On distingue quatre type de partitionnement :

- **Les partitions fixes**
  La quantité de ressources définissant une partition est statique et définie par l'administrateur lors de la configuration du gestionnaire de ressources et à l'avance de l'exécution des tâches. Toutes les tâches auront donc uniquement accès à des partitions équivalentes ;

- **Les partitions variables**
  La quantité de ressources définissant une partition variable est statique, elle est cependant définie d'après une requête de l'utilisateur ayant soumis la tâche. Cette requête spécifie des contraintes sur la partition que doit satisfaire le gestionnaire de ressources. Le Listing 2.1 présente une telle requête formulée pour le gestionnaire de ressources OAR [CDCG+05] ;

- **Les partitions adaptatives**
  Extension des partitions variables, le gestionnaire de ressources indique cependant à la tâche la liste des ressources qui lui ont été allouées lors de son démarrage. Les applications s'exécutant sur la grappe peuvent donc adapter le nombre de processus lancée à la quantité de processeurs allouée par exemple. Le calcul de la partition à allouer est réalisé lors du lancement de la tâche et peut être influencé par différentes règles ou selon l'état de charge de la grappe ;

- **Les partitions dynamique**
  Ce modèle de partition est le plus flexible. Dans ce contexte, l'environnement d'exécution permet d'allouer dynamiquement des ressources à une tâche [MVZ93, DGBL96]. Durant son exécution, l'application s'adapte d'elle même à la quantité de ressources disponibles.

```
1 fhermeni@front-sop:~$ oarsub -r "2009-05-11 18:00:00" \
      -l "{memnode=4096}/cluster=1/nodes=15/cpu=2/core=2,walltime=4:0:0"
```

Listing 2.1 – Requête réservant 15 nœuds d'une même grappe devant disposer chacun de 2 processeurs avec 2 cœurs et de 4096 Mo de mémoire vive pour 4 heures, à partir du 11 mai 2009 18h.

La Table 2.1 recense les différents types de partitions utilisables avec les différents modèles de tâches. Les partitions fixe et variables par exemple conviennent aux tâches rigides. Cependant, le partitionnement fixe à un cadre d'utilisation plus limité et convient seulement aux grappes n'exécutant qu'un seul type de tâche, connu à l'avance. Les partitions adaptatives conviennent aux tâches malléables tandis que

les partitions dynamiques conviennent aux tâches évolutives. Les partitions variables sont couramment utilisées dans les grappes exécutant des tâches ayant des besoins en ressources spécifiques. Les partitions adaptatives et dynamiques sont peu utilisées dans les grappes de serveurs. Elles nécessitent à la fois un environnement d'exécution capable d'allouer ou de désallouer dynamiquement des ressources aux processus et un intergiciel ou un modèle de développement permettant à l'application de s'adapter dynamiquement à la disponibilité des ressources.

Une solution à base de partitions variables mais permettant d'obtenir un résultat proche des partitions adaptatives consiste à soumettre au gestionnaire de ressources une tâche nécessitant la plus grande partition utile. Si le gestionnaire de ressources refuse cette soumission, alors une nouvelle requête est réalisée et demande une partition réduite. L'intérêt de cette solution est limité puisque le calcul de la partition se fait au moment de la soumission et non au moment du lançement de la tâche. Le Listing 2.2 représente une telle requête formulée pour le gestionnaire de ressources OAR.

```
fhermeni@front-sop:~$ oarsub -I -l nodes=4,walltime=2 \
    -l nodes=2,walltime=4
```

Listing 2.2 – Requête réservant de manière interactive 4 nœuds pour une durée de 2 heures ou en cas d'échec, 2 nœuds pour une durée de 4 heures.

Afin d'exécuter des tâches évolutives dans des conditions satisfaisantes, une solution consiste à surestimer les besoins en ressources avec une partition variable . L'utilisateur soumet sa tâche en spécifiant une partition de ressources correspondant au pire cas de l'application (quand chaque processus nécessite un processeur dédié, par exemple). La tâche s'exécutera alors dans les meilleurs conditions possibles mais réservera une quantité de ressources qui ne sera pas nécessairement utilisée.

Tâche	Partitionnement			
	Fixe	Variable	Adaptatif	Dynamique
Rigide	✓	✓		
Malléable		≈	✓	
Évolutive		≈		✓

Table 2.1 – Type de partition adapté aux différents modèles de tâches.

2.2.3 Type de soumission d'une tâche

Il est possible de soumettre une tâche de différentes façons :

- **Réservation**
  L'utilisateur choisit l'heure de lancement de la tâche en plus de la durée et de la quantité de ressources à réserver. Il est souvent nécessaire de préciser différents paramètres additionnels, comme le chemin vers un script à exécuter au démarrage ou à l'issu de l'exécution de la tâche. En fonction de la politique d'exécution du gestionnaire de ressources, la tâche peut être lancée à l'heure, en avance, ou dans de rare cas, après l'heure spécifiée par l'utilisateur. Dans ce mode, l'exécution de la tâche est entièrement sous le contrôle du gestionnaire de ressources ;

- **Exécution interactive**
  L'utilisateur ne spécifie pas de date de démarrage pour l'exécution de la tâche mais laisse le gestionnaire de ressources planifier l'exécution au plus tôt. Dans cette situation, le gestionnaire de ressources peut annoncer une estimation du temps d'attente avant l'exécution et si nécessaire, fournir un accès distant aux machines exécutant la tâche pour la durée de la réservation. Dans cette configuration, le gestionnaire de ressources ne fait qu'allouer des ressources. L'utilisateur peut les utiliser à sa guise, lancer différentes applications à la main ou préparer son environnement pour des exécutions ultérieures à base de réservations ;

- Le mode ***best-effort***
  Une tâche soumise en mode *best-effort* peut être exécutée dès que possible quand une partition valide devient disponible, mais le gestionnaire de ressources peut décider à tout moment d'arrêter la tâche si une tâche de plus grande priorité doit être exécutée (si une réservation est prévue par exemple). Si la tâche est préemptible alors le gestionnaire de ressources peut décider de la suspendre pour reprendre son exécution ultérieurement. Dans le cas contraire, la tâche sera simplement détruite. Ce mode peut être très utile lorsqu'il est nécessaire de lancer un grand nombre de tâches, avec des durées d'exécution courtes.

2.3 Ordonnancement des tâches

Une grappe de serveurs permet à plusieurs utilisateurs d'exécuter différentes tâches simultanément. Chaque tâche nécessite une certaine quantité de ressources pendant son exécution. Il est alors nécessaire de coordonner le partitionnement des ressources avec les besoins des différentes tâches. Un ordonnanceur a pour objectif de gérer de planifier l'exécution des tâches dans le temps. Il est donc chargé d'allouer à chaque tâche une partition de ressources permettant de l'exécuter dans des conditions suffisantes et si possible en réduisant au maximum le temps entre sa soumission et son exécution.

Le partitionnement des ressources doit tenir compte du niveau de flexibilité des tâches et des possibilités de partitionnement supportées par l'environnement et les tâches. Nous décrivons dans cette section les différents types d'ordonnancement, éventuellement combinables, permettant l'exécution en parallèle de plusieurs tâches. Deux types d'ordonnanceurs se dégagent de notre analyse, les ordonnanceurs reposant sur un partitionnement spatial qui proposent un partitionnement disjoint des ressources pour la totalité du temps d'exécution de chaque tâche et les ordonnanceurs à base de partitionnement temporel qui allouent des ressources aux tâches sous la forme de tranche de temps.

2.3.1 Partitionnement spatial

Le partitionnement spatial réserve pour chaque tâche une partition en veillant que l'ensemble de toute les partitions soit disjoint [SGS96]. Il n'y a donc pas de partage des ressources et l'allocation est faite pour la totalité du temps d'exécution de la tâche. Cette approche est habituellement utilisée avec des tâches rigides ou malléables et un partitionnement fixe ou variable.

L'algorithme d'ordonnancement doit décider d'un découpage des partitions le plus efficace possible afin d'exécuter simultanément un maximum de tâches en parallèle. La représentation graphique de ce partitionnement se fait généralement à l'aide d'un diagramme de GANTT. L'axe des abscisses sert d'indicateur temporel tandis que l'axe des ordonnées représente la liste d'un type de ressources disponible, habituellement la liste des processeurs. Chaque tâche en cours d'exécution est alors représentée par un ou plusieurs rectangles, permettant de repérer la liste des ressources allouées ainsi que la durée de l'allocation (voir Figure 2.1).

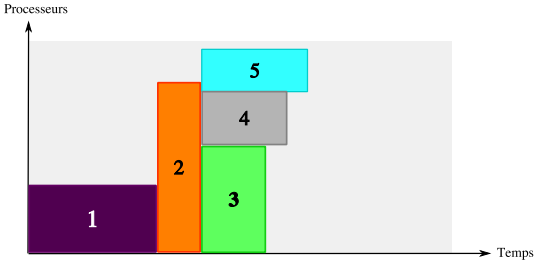


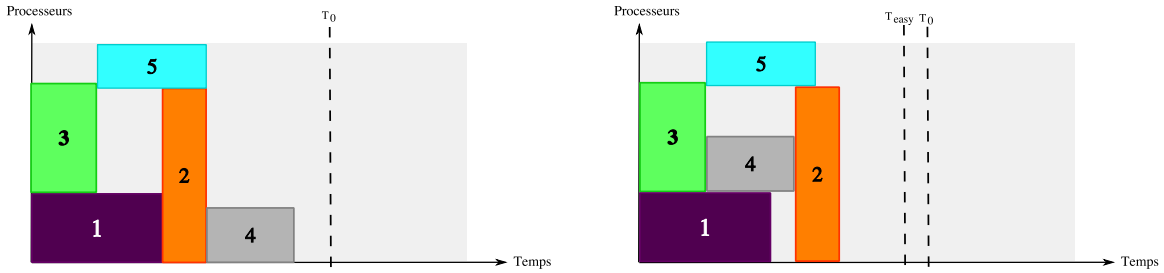
FIGURE 2.1 – Exemple d'un ordonnancement spatial de 5 tâches avec l'algorithme FCFS. L'algorithme exécute les tâches selon leur ordre d'arrivé, représenté ici par l'identifiant des tâches.

Les algorithmes d'ordonnancement spatial se basent sur un traitement par lots (*batch scheduling*) des tâches. Ils ont pour objectif d'exécuter une charge de travail, constituée de plusieurs tâches sur un ensemble de ressources bornées et sur une période de temps la plus réduite. Le moment de l'exécution

d'une tâche diffère le plus souvent du moment de sa soumission. Chaque tâche soumise est placée dans une file d'attente qu'un algorithme d'ordonnancement parcourt pour sélectionner les tâches à exécuter. Il existe différents algorithmes d'ordonnancement, chacun se basant sur des critères de sélection spécifiques.

Les algorithmes basés sur l'heure de soumission exécutent les tâches dans l'ordre où elles ont été soumises. L'algorithme *First Come, First Serve* (FCFS) consiste à exécuter chaque tâche dans un ordre strict correspondant à l'heure d'arrivée de la tâche dans la file. Dès qu'il est possible d'obtenir une partition suffisamment grande pour exécuter la première tâche dans la file, celle-ci est retirée de la file et est exécutée. La Figure 2.1 décrit un exemple d'allocation réalisé avec l'algorithme FCFS. La file d'attente contient à l'origine cinq tâches où chaque numéro correspond à son ordre d'arrivée. Pour exécuter la tâche 2, il est nécessaire d'attendre la fin de l'exécution de la tâche 1 afin d'obtenir une partition suffisante. Une fois la tâche 2 terminée, les tâches 3, 4 et 5 peuvent être lancées en parallèle puisqu'il y a suffisamment de processeurs libres pour les exécuter simultanément. L'algorithme FCFS est très simple à mettre en œuvre cependant ce séquencement strict des tâches ne permet pas d'occuper efficacement les ressources. En effet, si la première tâche de la file nécessite une grande quantité de ressources, elle peut bloquer inutilement la file durant un temps significatif, alors que plusieurs tâches nécessitant moins de ressources mais situées plus loin dans la file pourraient être exécutées.

Le *backfilling* propose d'utiliser un ordre non strict d'exécution des tâches afin de limiter la fragmentation des ressources. Cette stratégie tente d'occuper au mieux les ressources inutilisées lorsqu'une grande tâche en attente bloque la file. Les différents algorithmes réalisant cette opération se caractérisent par leur niveau d'agressivité, c'est à dire l'impact qu'ils ont sur la notion d'ordre stricte de la file. Afin de réaliser cet ordonnancement, il est nécessaire que le gestionnaire de ressources dispose d'une estimation du temps d'exécution de la tâche (le *walltime*). Cette estimation est fournit le plus souvent par l'utilisateur. L'algorithme *EASY Backfilling* [Lif95, SCZL96] compacte les différentes tâches en maintenant l'heure de démarrage de la première tâche. Cet algorithme augmente le taux d'occupation des ressources et peut exécuter une liste de tâche plus rapidement (voir Figure 2.2(a)) tout en permettant de garder un certain niveau de déterminisme sur le temps d'exécution des tâches. Les algorithmes de *backfilling* conservatifs [Wei98] proposent un ordonnancement garantissant qu'aucune tâche ne pourra être lancée en retard. Finalement, l'approche *First Fit*, cherche à placer n'importe quelle tâche au plus tôt. Le taux d'occupation des ressources est amélioré par rapport à l'algorithme EASY mais cette approche, très agressive, est susceptible de retarder certaines tâches et ne permet donc pas d'assurer qu'une tâche puisse être exécutée dans un intervalle de temps donné. La Figure 2.2(b) par exemple, montre une allocation des tâches basée sur un algorithme *First Fit*. L'heure de départ de la tâche 2 est retardée afin d'exécuter les tâches 3, 4 et 5 plus tôt.



(a) FCFS + *Easy backfilling* (b) FCFS + *First Fit*

FIGURE 2.2 – Exemples d'algorithmes de *backfilling*. T_0 et T_{easy} indiquent respectivement le temps nécessaire à l'exécution des 5 tâches avec un ordre de file strict (Figure 2.1) ou avec l'algorithme *EASY-Backfilling*.

Bien que les stratégies basées sur l'heure d'arrivée soient les plus courantes, il existe des stratégies reposant sur d'autres critères. Les algorithmes basés sur la date de fin d'exécution sont utilisés dans le cas d'exécution de tâches temps-réel où il est nécessaire d'assurer qu'une tâche sera exécutée dans le pire cas avant une date précise. Les algorithmes basés sur la priorité des tâches permettent d'associer un poids à chaque tâche, en fonction de son besoin en ressource par exemple, ou de son importance. En associant un poids élevé aux petites tâches, il est possible de réduire le temps d'attente moyen avant d'exécuter

une tâche. Les algorithmes équitables affectent à chaque utilisateur de la grappe un quota de ressources, les tâches des utilisateurs qui ont le moins utilisé la grappe reçoivent alors une plus grande priorité et peuvent être exécuter plus rapidement. L'algorithme *Earlier Deadline First* (EDF) [LL73] attribue une priorité élevée aux tâches devant finir le plus tôt, il ne permet pas cependant de gérer les situations où la grappe est surchargée. L'algorithme *Robust Earliest Deadline* (RED) [BS93] assure un service minimum en proposant deux type de classes pour les tâches. Les tâches appartenant à la classe critique doivent impérativement être terminées avant leur date limite, tandis que les tâches appartenant à la classe non-critique peuvent subir un retard si nécessaire en cas de surcharge. Ces tâches servent alors de tampon afin d'assurer la bonne exécution des tâches critiques. Les approches à base de priorité uniquement sont déconseillées, en effet les algorithmes de base ne peuvent prévenir un risque de famine sur la grappe, c'est à dire une situation qui n'assure pas qu'une tâche sera exécutée dans un temps borné.

2.3.2 Partitionnement temporel

Les algorithmes d'ordonnancement à base de partitionnement temporel allouent une partition à chaque tâche pour une tranche de temps bornée (un quantum). À la fin de chaque quantum, l'ordonnanceur réalise un changement de contexte : il suspend la tâche en cours d'exécution et réveille la tâche suivante. Cette approche est utilisée dans la majorité des systèmes d'exploitation modernes à l'échelle des processus et permet d'obtenir sur une machine avec uniquement un processeur un système simulant l'exécution en parallèle de plusieurs processus. Cette approche est cependant limitée dans les grappes lorsque les processus communiquent entre eux par le biais de messages et que le nombre de processus à exécuter est supérieur au nombre de processeurs. Si un processus en cours d'exécution est bloqué parce qu'il attend un message de la part d'un processus suspendu, alors celui-ci occupe inutilement des ressources.

Feitelson *et al.* [Fei90] proposent avec l'ordonnancement coopératif explicite (*gang scheduling*) d'utiliser des tâches préemptibles pour exécuter simultanément les différents processus d'une même tâche dans les grappes afin d'empêcher l'exécution de processus en attente d'exécution. Durant chaque quantum, une tâche potentiellement multi-processus est exécutée sur un ensemble disjoint de processeurs. Une fois son quantum terminé, l'algorithme d'ordonnancement sélectionne la prochaine tâche à exécuter et demande un changement de contexte global. La figure 2.3 représente un exemple d'exécution basé sur du *gang scheduling*. Chaque ligne représente un quantum et chaque colonne représente un type de ressource, dans le cas présent un processeur. On observe très rapidement que cette approche entraîne une fragmentation importante et donc un taux d'occupation des ressources faible.

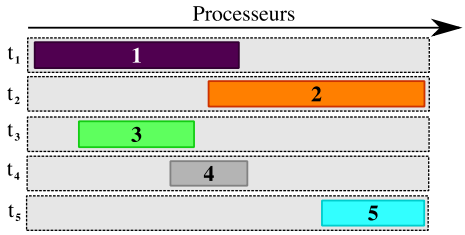
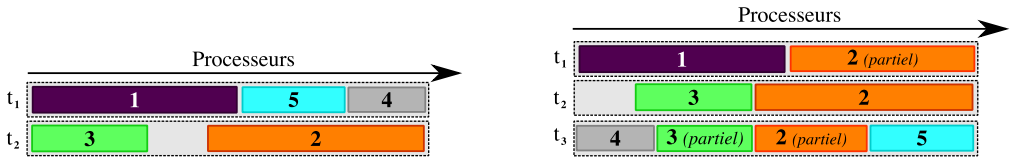


FIGURE 2.3 – Partitionnement temporel à base d'ordonnancement coopératif explicite (*gang scheduling*)

Concentrer plusieurs tâches sur un même quantum permet de limiter significativement la fragmentation [Fei96, FJ97]. Dans cette situation, à chaque fois qu'une nouvelle tâche est soumise, l'ordonnanceur vérifie s'il ne peut pas la placer sur un quantum déjà utilisé. S'il est possible de trouver une partition suffisamment grande alors la tâche s'exécutera avec les autres tâches du même quantum sinon, un nouveau quantum lui sera dédié. La Figure 2.4(a) montre ainsi qu'il est possible d'exécuter 5 tâches sur 2 quantums différents au lieu de 5 dans la Figure 2.3, les tâches s'exécuteront donc plus rapidement. Si l'environnement d'exécution et les applications supportent la migration de processus, il est également possible de migrer les processus durant leur ordonnancement pour diminuer la fragmentation à la volée et non seulement au moment de l'arrivée de nouveaux processus.

L'ordonnancement coopératif implicite (*co-scheduling*) propose une approche plus souple que le *gang scheduling* et permet d'exécuter les différents processus si nécessaire dans différents quantums. Cette approche relâche le souhait initial d'exécuter la totalité des processus d'une tâche dans un même quantum.



(a) Concentration des tâches

(b) Ordonnancement coopératif implicite (*co-scheduling*)

FIGURE 2.4 – Exemples d'approches réduisant la fragmentation dans le *gang scheduling*.

En réalité, la granularité proposée par le *co-scheduling* est plus fine et considère que l'inter-communication entre les processus n'est pas nécessairement uniforme. Il est donc possible de n'exécuter sur un même quantum que les processus les plus fortement liés. La désignation des processus fortement liés peut être faite par l'utilisateur lors de la description de sa tâche, ou réalisée à la volée par le gestionnaire de ressources en observant les différentes connexions réseaux entre les processus. La Figure 2.4(b) montre une telle exécution. Dans cet exemple, la fragmentation est réduite en exécutant une partie de la tâche 2 et de la partie 3 durant les quantum t_1 et t_3.

Le changement de contexte global est une fonctionnalité critique dans les ordonnanceurs à partitionnement temporel. Il faut en effet assurer un temps de changement de contexte le plus court possible puisque durant celui-ci, aucune tâche n'est exécutée. Il faut également assurer que la durée d'un quantum est grande par rapport à la durée d'un changement de contexte afin que la grappe soit utilisée principalement pour l'exécution des tâches et non leur manipulation. Ce temps de changement de contexte est également critique si des applications utilisent des connexions réseaux. Si l'application est suspendue trop longtemps, alors la connexion réseau peut être fermée par les autres participants (dans le cas de l'expiration du délai maximal d'attente de réponse dans le protocole TCP par exemple). Synchroniser tous les changements de contexte des différents processeurs d'une grappe est coûteux et les stratégies de *gang scheduling* souffrent de problèmes de passage à l'échelle. Le *co-scheduling* permet de réduire ce coût et d'améliorer le passage à l'échelle en ne réalisant le changement de contexte que sur un sous-ensemble de nœuds.

2.4 Conclusion

Nous avons réalisé dans ce chapitre un état de l'art relatif à la gestion des tâches dans les grappes de serveurs. Les travaux portant sur les modèles de tâches prennent en compte la flexibilité à la fois des tâches et du mode de partitionnement des ressources pour proposer différents contextes d'utilisations. Les modèles totalement rigides considèrent une allocation et une utilisation des ressources constante tandis que les modèles dynamiques permettent un partitionnement des ressources adapté à des tâches ayant des besoins variables. Ces différents modèles servent de base à différentes stratégies d'ordonnancement permettant l'exécution simultanée de plusieurs tâches grâce à un découpage spatial ou temporel des partitions. Un des objectifs des gestionnaires de ressources est de permettre un taux d'occupation maximale des ressources afin d'exécuter les tâches au plus tôt. Les approches les plus efficaces proposent une gestion des tâches à la volée où les ressources sont allouées selon le besoin réel des applications et la charge de la grappe et non selon une estimation, souvent peu précise, des utilisateurs. De plus, le partitionnement temporel permet de gérer les tâches avec un grain plus fin en exécutant les tâches de manière préemptible.

L'implémentation de stratégies d'ordonnancement complexe tel que le *gang scheduling* ou le *co-scheduling* est plus délicate que l'implémentation de stratégies à base de partitionnement spatiale. Les stratégies à base de partitionnement temporelle repose en effet sur une gestion dynamique des tâches et nécessitent donc des mécanismes dédiés à la manipulation de celles-ci en temps réelles tels que la migration et la suspension sur disque. Proposer une implémentation générique de ces mécanismes avec un système d'exploitation standard et des processus est complexe [MDP+00] et tend à limiter l'utilisation de tels algorithmes [ET05].

Nous discuterons plus en détail des limitations actuelles de ces approches dans le chapitre 5 et nous proposeront une solution consistant à embarquer les composants des tâches dans des machines virtuelles

systèmes. Le chapitre suivant présente un état de l'art relatif à la virtualisation et aux machines virtuelles systèmes.

Chapitre 3

Les machines virtuelles

Où nous présentons un état de l'art sur la virtualisation système. Nous discutons des besoins historiques qui ont motivé le principe de protection dans les systèmes d'exploitation puis l'isolation par la virtualisation système. Nous discutons également des différentes approches pour la virtualisation système et des mécanismes de migration, de suspension et de reprise d'activité qui permettent de manipuler les machines virtuelles.

Sommaire

L'implémentation de stratégies d'ordonnancement complexes et dynamiques tel que le *gang scheduling* avec concentration des tâches nécessitent que leur support d'exécution mettent à disposition des mécanismes permettant de capturer l'état d'une tâche sur disque ou en mémoire, de la migrer sur un autre hôte ou de modifier dynamiquement la taille de sa partition.

Historiquement, le support d'exécution est un système d'exploitation embarquant chaque composant d'une tâche dans un processus. La pagination et la segmentation de la mémoire ainsi que les niveaux de privilège des instructions processeurs fournissent une protection suffisante. Cependant, de par sa conception, ce support implique que l'environnement d'exécution de la machine puisse exécuter l'application. Celle-ci peut cependant avoir été développée pour un système incompatible avec l'environnement de la machine (architecture matérielle, système d'exploitation, ...). De plus, une application embarquée dans un processus est liée à des objets tiers comme des bibliothèques, des descripteurs de fichier ou des connexions réseau qu'il est difficile de manipuler directement.

Contrairement aux systèmes d'exploitation basés sur le partage contrôlé des ressources, la virtualisation système repose sur l'isolation. La virtualisation est apparue dans les années 60 comme une solution permettant à plusieurs développeurs de travailler simultanément sur une même machine en isolant chaque personne sur sa propre instance qui reprend trait pour trait le fonctionnement de la machine physique. Ainsi une erreur de manipulation dans une instance ne pourra pas compromettre les autres instances. Avec l'arrivée de machines personnelles à faible coût, l'intérêt de la virtualisation a baissé. Cette infrastructure a cependant retrouvée un intérêt au début des années 2000 afin de concentrer des applications

peu gourmandes en ressources et s'exécutant sur différents systèmes sur un nombre limité de machines afin de réduire les coûts de fonctionnement et d'entretien des machines.

Nous dressons dans ce chapitre un état de l'art concernant les machines virtuelles et plus spécialement les machines virtuelles système. Après une brève description de la protection dans les systèmes d'exploitation, nous décrivons les concepts de la virtualisation applicative puis les concepts de la virtualisation système tel qu'ils ont été définis par Popek et Goldberg [PG74]. Nous présentons ensuite les différentes approches de la virtualisation système : la virtualisation pure logicielle ou assisté par le matériel, la para-virtualisation utilisant des systèmes d'exploitation préparés et enfin la virtualisation des systèmes d'exploitation. Nous décrivons enfin les mécanismes de capture d'état et de migration pour les machines virtuelles.

3.1 Protection dans les systèmes d'exploitation

Un des principes de base fournis par les systèmes d'exploitation est la protection. Cela consiste en un ensemble de mécanismes empêchant entre autre qu'une faute dans un programme ne puisse mettre d'autres programmes en faute. Les premiers systèmes ne proposaient pas de mécanismes pour limiter les espaces d'adressage par exemple et manipuler un pointeur sur sur une zone mémoire non-allouée dans une application pouvait faire échouer une autre application voir même le système entier. La protection de la mémoire est réalisée en partie par la segmentation [SS75] et la pagination. Ces approches bornent les plages mémoire accessibles à un processus afin d'empêcher les applications de manipuler la mémoire dont elles ne sont pas propriétaires. La protection est également présente au niveau des processeurs en limitant les instructions utilisables par une application. Chaque processeur met à la disposition des développeurs du système d'exploitation un jeu d'instructions appelé ISA. Certaines de ces instructions sont privilégiées et ne doivent être exécutables que par le système d'exploitation. Elles manipulent entre autre l'état du processeur et de la mémoire et une mauvaise utilisation de celles-ci peut compromettre le système. En 1972, le système Multics [SS72] propose d'une architecture disposant de huit niveaux de privilèges pour l'exécution des instructions processeur. Au niveau le plus bas, le mode superviseur, un noyau disposant de tous les privilèges exécute du code supposé sûr assurant les tâches les plus sensibles du système d'exploitation comme la gestion de la mémoire ou la communication avec les périphériques. Dans les niveaux plus élevés, tel que le niveau correspondant au mode moniteur, le code n'est pas considéré comme sûr et les applications disposent de privilèges réduits. Elles peuvent cependant utiliser la mémoire et les périphériques par le biais d'interfaces communiquant avec le noyau. Si une erreur se produit dans le mode moniteur, le fonctionnement de la machine n'est pas compromis et si une application souhaite exécuter des instructions réservées au mode superviseur, une exception liée à la sécurité est levée.

La plupart des systèmes actuels (UNIX, GNU/Linux, Windows) dérivent de Multics. Aujourd'hui les processeurs de type x86 proposent en mode protégé quatre niveaux d'exécution. Sous le noyau GNU/Linux, 2 niveaux sont réellement utilisés. Le niveau 0 (le mode superviseur) réservé au noyau et pouvant exécuter directement n'importe quelle instruction de l'ISA et le niveau 3 (mode moniteur) pour les applications des utilisateurs (voir Figure 3.1) et ne pouvant exécuter directement que des instructions non-privilégiées.

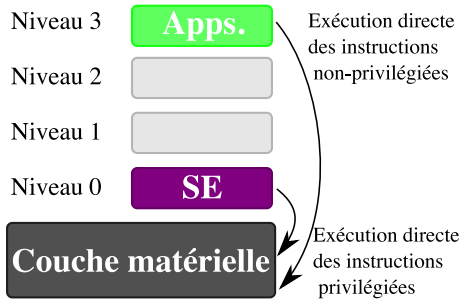


FIGURE 3.1 – Niveaux de privilèges pour les processeurs x86 hébergeant un système GNU/Linux

3.2 Machine virtuelle applicative

Dans le domaine des langages de programmation, une machine virtuelle décrit un environnement d'exécution comprenant un interpréteur ou un compilateur de code à la volée (appelé compilateur JIT) pour un langage de programmation. L'environnement Java, incluant un langage et une machine virtuelle [LY99] (la JVM) en est un exemple. Habituellement, le point d'entrée d'une machine virtuelle applicative n'est pas du code machine compatible avec l'architecture matérielle comme pour la virtualisation système, mais un pseudo code pour une machine abstraite. Ce pseudo code est alors interprété ou compilé à la volée pour être émulé sur la machine physique. Cette approche permet une grande portabilité des applications puisque le pseudo code de la machine abstraite peut être émulé avec différentes ISA afin d'être exécutable sur différents systèmes. Les performances obtenues avec de l'interprétation sont inférieures aux performances observables lors d'une exécution native cependant, l'utilisation de techniques de compilation JIT permet une amélioration notable.

3.3 Machine virtuelle système

Les approches dérivées de Multics proposent un partage contrôlé des ressources et de l'information. Les machines virtuelles au contraire sont construites autour de la notion d'isolation. Ce concept apparaît dans les années 60 comme une solution pour faciliter le travail des développeurs. Chacun travaille sur une instance différente de la machine, isolée des autres, qui recopie fidèlement l'architecture matérielle de la machine. Une erreur dans une instance ne peut alors se propager dans les autres instances. L'application responsable de la gestion et de la coordination des machines virtuelles est appelée le moniteur de machines virtuelles, ou hyperviseur. Cette application s'exécute avec un niveau de privilège plus élevé que le système d'exploitation, le mode *hyperviseur*. En 1974, Popek et Goldberg [PG74] définissent une machine virtuelle comme une réplique isolée et efficace de la machine réelle hôte satisfaisant les 3 propriétés suivantes :

- **Équivalence**
  L'exécution d'une application dans une machine virtuelle ou sur une machine physique doit être identique. Cette propriété exclut cependant la gestion du temps et la disponibilité des ressources ;
- **Contrôle**
  Les applications fonctionnant dans une machine virtuelle ne doivent pas avoir accès aux ressources physiques de la machine sans accord préalable de l'hyperviseur ;
- **Efficacité**
  Les instructions non-privilégiées s'exécutant dans la machine virtuelle doivent être traitées directement par le processeur.

Dans le contexte de la virtualisation, il est possible de classifier les instructions des processeurs à virtualiser dans 2 catégories :

- **Les instructions non-sensibles**
  Elles ne remettent pas en cause la protection et l'isolation. Ces instructions peuvent donc être exécutées directement par le processeur, au niveau moniteur, sans traitement préalable de l'hyperviseur.
- **Les instructions sensibles**
  Elles remettent en cause l'isolation ou la protection des machines virtuelles. On y trouve entre autre les instructions consultant ou modifiant l'état de la machine, les instructions liées aux entrées/sorties et les instructions se comportant différemment en fonction du niveau d'exécution du processeur.

3.4 Les approches pour la virtualisation système

La virtualisation système a été implémentée selon différentes approches. Elles se différencient principalement par leur respect des propriétés définies par Popek et Goldberg, par l'utilisation de systèmes d'exploitations « préparés » ou par l'utilisation de matériel spécifique.

3.4.1 Virtualisation pure

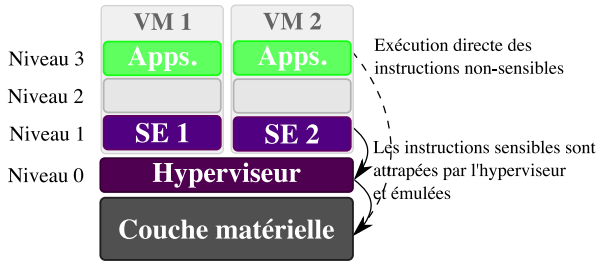


FIGURE 3.2 – Virtualisation pure logicielle

La virtualisation pure permet l'exécution de machines virtuelles sans modification des systèmes d'exploitation et permet une émulation de chaque instruction sensible. Popek et Goldberg ont énoncé qu'un processeur était virtualisable de façon pure si l'ensemble de ses instructions sensibles est un sous-ensemble des instructions privilégiées. Il est alors possible de détecter les instructions sensibles en exécutant les machines virtuelles en mode moniteur (voir Figure 3.2). Dans ce cas, l'exécution d'une instruction sensible dans une machine virtuelle lèvera une exception de sécurité. L'exception sera attrapée par l'hyperviseur qui pourra alors l'émuler de façon sûre. Le système VM/370 [Cre81], proposé par IBM en 1981, est un exemple historique d'une telle approche.

La virtualisation pure a cependant des limites. Il faut en effet disposer d'une infrastructure système virtualisable, comme ce fut le cas pour les machines VM/370. Les processeurs de type x86 quant à eux contiennent 17 instructions sensibles pouvant fonctionner en mode non-privilégié [RI00]. Ces processeurs ne sont donc pas virtualisables. L'exécution de ces instructions en mode moniteur ne déclenche pas d'exceptions, l'hyperviseur ne peut donc pas les intercepter. La solution proposée par l'hyperviseur VMWare [SVL01] consiste à réécrire à la volée certaines sections de code. Lorsqu'une instruction sensible est détectée, VMWare réécrit le code binaire émulant cette instruction et le place dans un cache afin de ne traduire la section qu'une seule fois. Attraper les exceptions pour les émuler est un processus coûteux impliquant de fréquents changements de contexte. La virtualisation pure a donc un impact significatif sur les performances. Cet impact est dû à la fois aux systèmes d'exploitation qui ne sont pas nécessairement adaptés à la virtualisation mais également aux processeurs qui ne sont pas nécessairement virtualisables.

3.4.2 Para-virtualisation

La virtualisation pure permet l'exécution native des systèmes d'exploitation, mais entraîne une baisse significative des performances. En effet, l'émulation stricte de toutes les instructions sensibles est coûteuse et il serait préférable de proposer dans certaines situations une bibliothèque permettant de mieux gérer certains besoins des machines virtuelles, par exemple les entrées/sorties ou la mise à jour des tables de pages mémoire. Lorsque le code source du système d'exploitation est disponible, il est possible de réaliser ces changements. Cette approche permet d'améliorer les performances mais a pour inconvénient de ne pas respecter strictement la règle d'équivalence de Popek et Goldberg.

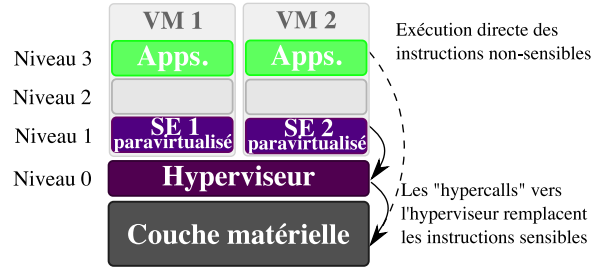


FIGURE 3.3 – Paravirtualisation avec l'hyperviseur Xen

La para-virtualisation limite l'impact de la virtualisation sur les performances. Cette approche consiste à modifier à l'avance le code des systèmes d'exploitation des machines virtuelles. Ce procédé a été décrit originalement dans l'hyperviseur Denali [WSG02] puis dans Xen [BDF+03]. Les appels de fonctions produisant des instructions sensibles ou des opérations optimisables sont remplacés par des appels directs à l'hyperviseur (des *hypercall*), ce qui supprime la nécessité d'attraper des exceptions à la volée (voir Figure 3.3). Cette approche a permis d'optimiser le code des systèmes d'exploitation paravirtualisés. Dans Denali par exemple, une nouvelle instruction *idle-with-timeout* permet d'éviter qu'une machine virtuelle ne gaspille trop de cycles horloge en attente active et l'hyperviseur Xen autorise un accès direct en lecture à la mémoire de chaque machine virtuelle afin de limiter les changements de contexte dus au passage en mode l'hyperviseur.

3.4.3 Virtualisation pure assistée

La virtualisation pure assistée se base sur les dernières générations de processeurs Intel [NSL+06] et AMD [AMD05]. L'ISA a été modifiée pour rendre les processeurs à nouveau virtualisables et de nouvelles instructions proposent de gérer le changement de contexte entre les machines virtuelles de manière matérielle. Les processeurs proposent également un nouveau niveau d'exécution dédié à l'hyperviseur. Les systèmes d'exploitation des machines virtuelles peuvent alors de nouveau s'exécuter au niveau superviseur (voir Figure 3.4).

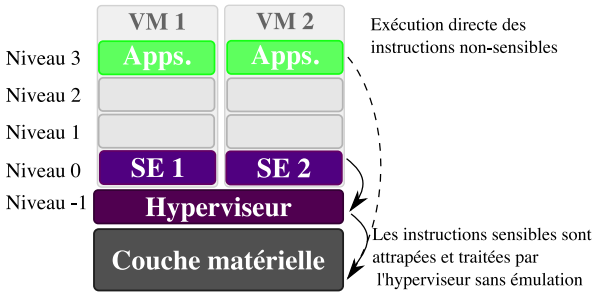


FIGURE 3.4 – Virtualisation pure assisté

La virtualisation pure assistée permet en théorie d'obtenir de meilleures performances que les approches à base de traduction binaire ou de para-virtualisation en apportant des mécanismes propres à la gestion des machines virtuelles de manière matérielle. Dans la pratique, ce gain est mitigé et dépendant de la charge des machines virtuelles [AA06]. Si la machine virtuelle crée beaucoup de processus, réalise des entrées/sorties ou des changements de contexte fréquemment, l'approche purement logicielle sera plus performante qu'une approche à base de virtualisation assistée. Ce surcoût s'explique en partie par les changements de contexte fréquents que doit réaliser le processeur afin de passer du mode superviseur au mode hyperviseur alors que ce changement de contexte n'est pas nécessaire avec une approche purement logicielle.

3.4.4 Virtualisation du système d'exploitation

Dans certaines situations, il n'est pas nécessaire de virtualiser un système entier. Par exemple, si l'on souhaite isoler fortement différentes applications fonctionnant sur un même système d'exploitation, fournir une virtualisation de la machine physique n'est pas nécessaire mais il faut cependant assurer une forte isolation entre les différentes applications pour des questions de sécurité ou de contrôle des ressources. L'appel système *chroot()* change la racine du système pour un processus donné et réalise ainsi une prison logicielle en chargeant celui-ci et ses dépendances dans un environnement cloisonné. Cependant le contrôle des ressources est limité et la sécurité n'est pas garantie, un simple accès super-utilisateur permet de casser le cloisonnement par exemple.

La virtualisation du système d'exploitation [SPF+07, hKW00, PT04] permet de faire fonctionner sur un seul système d'exploitation plusieurs *conteneurs* en proposant la virtualisation au niveau des appels système du système d'exploitation de l'hôte. La couche dédiée à la virtualisation peut être implémentée

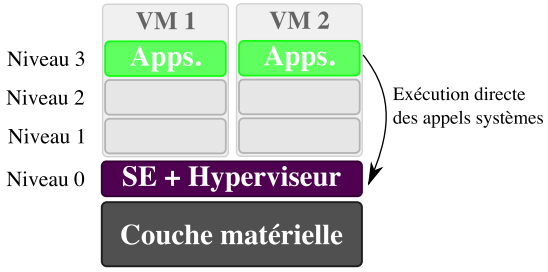


FIGURE 3.5 – Virtualisation du Système d'Exploitation avec VServer

au dessus du système d'exploitation de la machine ou faire partie de celui-ci comme c'est le cas pour VServer [SPF+07] (voir Figure 3.5). Chaque machine virtuelle dispose de son propre système de fichiers, les espaces de nommage et d'adressage mémoire sont totalement disjoints comme pour la virtualisation système, le contrôle des ressources et l'isolation des périphériques sont réalisés en utilisant des algorithmes d'allocation et d'ordonnancement équitables. Par rapport à une protection basée sur l'appel système *chroot()* et les processus, la sécurité et l'isolation sont renforcées et le contrôle sur les ressources est amélioré.

La virtualisation du système d'exploitation ne nécessite pas d'émuler ou de traduire les appels système et permet donc d'obtenir des performances proches d'une exécution native. L'isolation des machines virtuelles et le contrôle des ressources sont réalisés par des mécanismes plus légers que dans le cas de la virtualisation système et permettent un meilleur passage à l'échelle en supportant, sur une même machine physique, beaucoup plus de machines virtuelles. Son principal inconvénient est son manque de flexibilité dû à l'obligation d'utiliser, dans toutes les machines virtuelles, le système d'exploitation de la machine hôte.

3.5 Hyperviseur natif, hyperviseur applicatif

Les hyperviseurs VM/370, Xen et VMWare ESX Server, par exemple, sont des hyperviseurs natifs (appelés hyperviseurs de Type I). Ils s'exécutent au dessus de la couche matérielle. Ces hyperviseurs sont des systèmes d'exploitation à proprement dit, mais dont le seul rôle est la gestion des machines virtuelles. Cette solution implique que les pilotes mis à disposition des machines virtuelles seront fournis par l'hyperviseur. Le niveau de sécurité est supposé meilleur puisque l'hyperviseur n'utilise pas de composants extérieurs. Cette approche permet également d'obtenir de meilleures performances puisque les pilotes sont spécialisés pour l'hyperviseur. Ce niveau d'implémentation limite par contre le développement et le déploiement des machines virtuelles puisqu'il est nécessaire de fournir les pilotes qui seront utilisés par les systèmes d'exploitation des machines virtuelles.

Les approches à base de virtualisation du système d'exploitation et les hyperviseurs KXen, VMWare Fusion et VMWare Workstation sont des hyperviseurs logiciels (hyperviseurs de Type II). Ils s'exécutent au dessus du système d'exploitation de la machine. Cette approche permet d'utiliser les pilotes fournis par le système d'exploitation et facilite ainsi le développement et le déploiement des machines virtuelles. En contre-partie, les performances sont amoindries : l'hyperviseur s'exécute en concurrence avec d'autres applications et utilise des pilotes qui peuvent être peu adaptés à la virtualisation.

3.6 Capture d'état et migration des machines virtuelles

Dans les grappes de serveurs, des outils de capture d'état ou de migration de processus ont été développés afin de réaliser à la volée de l'équilibrage de charge, de la haute disponibilité ou de la reprise de calcul en cas de fautes. Beaucoup de solutions ont été développées afin de proposer des mécanismes plus efficaces, plus rapides et les moins invasifs possible [HD06a]. Ces solutions imposent cependant un environnement précis, qui n'est pas nécessairement compatible avec l'environnement d'exécution des tâches des utilisateurs. Une application s'exécutant dans un processus dispose de dépendances avec des bibliothèques, des descripteurs

de fichier ouverts ou des connexions réseau qui doivent être prisent en compte durant la capture d'état ou la migration. La détection et la gestion de ces dépendances résiduelles a limité le déploiement de telles solutions [MDP+00]. L'utilisation des machines virtuelles comme composant de base permet de résoudre en partie ces problèmes. Une machine virtuelle est en effet complètement isolée et contient toutes les bibliothèques dont dépendent les applications. L'hyperviseur a la connaissance de tous les liens entre la machine virtuelle et l'environnement extérieur et est indépendant de tout environnement logiciel utilisateur.

Capturer l'état d'une machine virtuelle est utile entre autre pour faire de la reprise d'exécution en cas de faute. Pour les applications de calcul haute performance par exemple, si une erreur survient durant l'exécution, il est possible de relancer la machine virtuelle depuis sa dernière sauvegarde plutôt que depuis son lancement initial. Migrer une machine virtuelle est une extension de cette action et implique que la machine réalisant la capture soit différente de la machine physique restaurant la machine virtuelle. La capture d'état et la migration peuvent toutes deux être exécutées à froid, en suspendant la machine virtuelle durant tout le processus, ou à chaud en suspendant l'activité de la machine virtuelle le minimum de temps possible. Ces deux actions sont semblables et les mécanismes associés sont identiques.

Lors d'une capture d'état ou d'une migration à froid d'une machine virtuelle, l'hyperviseur utilise le principe de *stop-and-copy* [KS02, SCP+02] pour suspendre la machine virtuelle dans un état cohérent. L'hyperviseur désactive les interruptions puis bloque les processeurs virtuels. Le contexte d'exécution de la machine virtuelle et sa mémoire sont ensuite écrits dans un fichier dans le cas d'une capture d'état ou transmis sur la machine destination dans le cas d'une migration. La restauration de la machine virtuelle se fait en répétant l'opération dans le sens inverse. Cette approche permet de suspendre rapidement la machine virtuelle en ne traitant qu'une seule fois chaque page mémoire et la durée du processus dépend principalement de la quantité de mémoire allouée à la machine virtuelle. Durant tout ce processus, la machine virtuelle est indisponible. Si cette durée est longue, il n'est pas possible d'exécuter ces opérations sur des machines virtuelles exécutant des services haute disponibilité. De plus, si la machine virtuelle dispose de connexions réseau avec d'autres machines, un temps d'indisponibilité trop grand risque d'empêcher leur restauration.

La capture d'état et la migration à chaud réduisent le temps d'indisponibilité de la machine virtuelle au minimum. Les approches proposées dans Xen [CFH+05] et dans VMWare [NLH05] reposent sur le principe de *pré-copie*. Alors que la machine virtuelle reste active sur la machine hôte, le contenu de sa mémoire est stocké dans un fichier ou transféré sur la machine destination en plusieurs passes. À chaque passe, les pages mémoire non-copiées où dont le contenu a changé sont copiés. Après plusieurs itérations, la machine virtuelle est suspendue sur l'hôte, son état processeur ainsi que les dernières pages mémoires sont copiées. Une fois cette dernière phase de copie réalisée, la machine virtuelle est restaurée. Cette approche entraîne un temps de traitement plus long puisque les pages peuvent être copiées plusieurs fois, mais maintient un temps d'indisponibilité court et indépendant de la taille mémoire allouée à la machine virtuelle. La migration par *post-copy* [HG09] suspend d'abord la machine virtuelle afin de transférer son état processeur, restaure la machine virtuelle sur la machine destination et transfère les pages mémoires ensuite. Cette approche maintient un temps d'indisponibilité relativement réduit, et permet d'obtenir un temps total de migration proche d'une approche par *stop-and-copy* puisque chaque page n'est transmise qu'une fois.

Une difficulté supplémentaire de la migration à chaud est le maintien des connexions réseau ou des accès aux données lors du changement d'hôte. Pour maintenir les connexions réseau, Xen transfert l'adresse IP de la machine virtuelle lors de la migration puis signale sa nouvelle position en envoyant des réponses ARP sur le réseau. Certains paquets IP peuvent être perdu durant la migration cependant les protocoles réseaux sont sensés pouvoir supporter cette perte. Cette approche a des limites lorsque la machine virtuelle doit être migrée entre deux machines appartenant à deux réseaux distants. En effet, les routeurs ne laissent pas passer les requêtes ARP, il faut alors prévoir des mécanismes de re-routage [HGM+07] ou utiliser une solution à base de Mobile IP [TDG+06], incorporée en standard dans le protocole IPv6. La gestion des données est par contre ignorée dans Xen qui considère que les données sont partagées et accessibles de façon sûre sur la machine source et la machine destination, par le biais de serveurs de fichiers ou de réseaux de stockage par exemple.

Stocker l'image d'une machine virtuelle sur disque lors d'une capture d'état est coûteux, à la fois en espace disque et en temps. Il faut en effet écrire sur le disque au moins la totalité de la mémoire vive

allouée à la machine virtuelle. Pour réduire ce temps, il est possible d'utiliser par exemple des mécanismes de *copy-on-write*. Le système de fichier distribué Parallax [MAC+08] par exemple, a été implémenté pour supporter des captures d'état très fréquentes. Pour cela, le processus de capture d'état est incrémental et n'écrit sur le disque que les changements ayant eu lieu depuis la dernière sauvegarde.

3.7 Conclusion

Nous avons dans ce chapitre dressé un état de l'art des mécanismes de virtualisation système. la virtualisation permet d'inclure dans un composant, une machine virtuelle, l'environnement d'exécution d'une application. Un hyperviseur assure ensuite leur exécution et permet un contrôle fin de l'utilisation de leurs ressources. L'isolation fournit par les hyperviseurs permet de faire fonctionner les environnements des utilisateurs sans nécessairement impliquer de ré-ingéniérie. De plus, l'implémentation des mécanismes de capture d'états et de migrations de machines virtuelles au niveau de l'hyperviseur permet de les manipuler facilement et de façon transparente. Nous avons discuté de différentes approches permettant de réaliser de la virtualisation systèmes : alors que la virtualisation pure et assistée permettent d'exécuter des systèmes d'exploitation sans ré-ingénierie, la para-virtualisation permet d'exécuter des systèmes d'exploitation préparés pour la virtualisation.

En contrepartie, un environnement utilisant des machines virtuelles est plus lourd à manipuler. La gestion du stockage des images disques des machines virtuelles est contraignante si l'on souhaite utiliser les fonctionnalitées de migration et nécessite du matériel spécialisé (un réseau de stockage par exemple) pour être performante. Cependant, différentes approches basées sur l'observation des cas d'utilisations des machines virtuelles apparaissent actuellement et devrait apporter des solutions viables facilitant le stockage des images disques de celles-ci. La virtualisation des systèmes a également un impact sur les performances qui peut être significatif dans certains cas. Les mécanismes de gestion de mémoire ou des entrées/sorties des hyperviseurs actuelles peuvent par exemple rendre leur utilisation trop pénalisante pour du calcul haute performance. Il conviendrait alors de développer des hyperviseurs spécialisés pour un matériel précis et exécutant des systèmes d'exploitations optimisés afin de réduire cette dégradation. Finalement, la virtualisation matérielle des processeurs x86 a permis de renouveler l'intérêt de la virtualisation pure permettant d'exécuter des systèmes d'exploitations nativement. Cette approche, encore jeune, implique cependant un impact significatif sur les performances et rivalise difficilement avec une virtualisation système purement logicielle ou avec la para-virtualisation.

Nous pouvons réaliser un parallèle entre les fonctionnalités dédiées à la manipulation des machines virtuelles et les fonctionnalités manipulant les processus dans les stratégies d'ordonnancement réalisant une gestion dynamique des tâches. Nous reviendrons en détail sur l'intérêt d'utiliser la virtualisation dans les grappes pour l'ordonnancement de tâches dans le chapitre 5. Dans le chapitre suivant, nous réalisons un état de l'art sur la programmation par contraintes, une méthode flexible pour la définition et la résolution de problèmes combinatoires incluant les problèmes d'ordonnancement et de placement tel que ceux traités par les gestionnaires de ressources

Chapitre 4

La Programmation Par Contraintes

Où nous décrivons les principes de base de la Programmation Par Contraintes, approche retenue pour la résolution de problème de placement et d'ordonnancement. Nous insistons sur la flexibilité de cette approche déclarative, capable de résoudre des problèmes combinatoires variés qu'il est possible de composer en ajoutant au besoin des conditions que les solutions doivent satisfaire.

Sommaire

Un gestionnaire de ressources a pour objectif d'orchestrer l'exécution des différentes tâches qui lui ont été soumises. Dans la pratique, un ou plusieurs algorithmes sélectionnent les tâches à exécuter parmi l'ensemble des tâches soumises. Pour chacun des composants des tâches sélectionnées, le gestionnaire de ressources recherche une place sur un nœud de calcul. Chacun de ces algorithmes est dédié à la satisfaction de critères spécifiques aux tâches mais également à la grappe et aux objectifs de l'administrateur.

Les problèmes de sélection des tâches à exécuter ainsi que l'affectation de leur composant à des nœuds correspond respectivement à des problèmes d'ordonnancement et de placement. Si l'on souhaite ne pas limiter la capacité d'adaptation du gestionnaire de ressources à des problèmes inadaptés à certaines situations, il importe alors de sélectionner une approche flexible pour la définition et la résolution de ces problèmes.

Les problèmes d'ordonnancement et de placement forment une classe de problèmes d'optimisation combinatoire généralement complexe. Différentes approches de résolution existent et reposent sur des algorithmes génériques, par exemple la Programmation Linéaire, SAT ou la Programmation Par Contraintes [RvBW06] (PPC). La Programmation Par Contraintes est une approche récente, apparue dans la fin des années 1980. Elle consiste à modéliser un problème par un ensemble de relations logiques, des contraintes, imposant des conditions sur l'instantiation possible d'un ensemble de variables décrivant une solution du problème à des valeurs. Un solveur de contraintes calcule une solution en instantiant chacune des variables à une valeur satisfaisant simultanément toutes les contraintes. La PPC est une approche déclarative où l'utilisateur décrit un problème en le décomposant dans un langage de haut niveau. Contrairement aux autres approches génériques de résolution telles que la Programmation Linéaire ou

SAT, la PPC permet de formuler de manière explicite et flexible des problèmes combinatoires complexes. Dans le cadre de cette thèse, nous avons choisi d'utiliser une approche à base de PPC pour la définition et la résolution de nos problèmes d'ordonnancement et de placement. Nous justifierons plus en détail ce choix dans le Chapitre 5.

Nous discutons dans ce chapitre des principes fondateurs de la PPC. Dans un premier temps nous décrivons le modèle de déclaration des problèmes de satisfaction de contraintes et des problèmes d'optimisation sous contraintes. Dans une seconde partie, nous discutons des méthodes de résolution des problèmes à base de contraintes et des solutions permettant d'accélérer le processus de résolution. Finalement, nous présentons les principes des solveurs de contraintes qui mettent à disposition des utilisateurs une bibliothèque de contraintes génériques pour la modélisation et la résolution de problèmes variés.

4.1 Modélisation d'un problème à base de contraintes

Nous définissons dans cette section les différents éléments permettant de modéliser un problème combinatoire par une conjonction de contraintes. Dans une première partie, nous définissons les problèmes de satisfaction de contraintes puis nous présentons les problèmes d'optimisation sous contraintes dont les solutions maximisent ou minimisent la valeur d'une fonction de coût donnée.

4.1.1 Les problèmes de satisfaction de contraintes

Une contrainte est une relation logique qui est établie entre différentes inconnues appelées variables. Chaque variable prend sa valeur parmi un ensemble qui lui est propre appelé domaine. Définir une contrainte sur un ensemble de variables consiste à interdire l'instantiation simultanée de variables à certaines combinaisons de valeurs. Par exemple, la contrainte $(x, y) \in \{(0,1), (1,2)\}$ précise que la variable x ne peut prendre la valeur 0 que si la variable y prend la valeur 1 et que la variable y peut prendre la valeur 2 uniquement si x prend la valeur 1. Une telle contrainte est définie en extension, c'est à dire en spécifiant explicitement les tuples de valeurs qu'elle autorise (ou interdit). Une contrainte peut également être définie en intention par une équation mathématique (ex : $x, y \in [0,2] \mid x + y \leq 4$), une relation logique simple ($x \leq 1 \Rightarrow y \neq 2$) ou même par un prédicat portant sur un nombre de variables indéfini (ex : allDifferent $(x_1, \ldots, x_2)$). Une telle contrainte est appelée contrainte globale. Nous reviendrons sur la définition et l'intérêt des contraintes globales dans la section 4.2.3.

Un problème de satisfaction de contraintes (CSP) est défini formellement par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ représentant :

- un ensemble fini de variables $\mathcal{X}$;
- une fonction $\mathcal{D}$ affectant à chaque variable $x_i \in \mathcal{X}$ son domaine $\mathcal{D}(x_i)$, c'est à dire l'ensemble de valeurs que peut prendre cette variable. Dans un CSP discret, on suppose que $\mathcal{D}(x_i) \in \mathbb{Z}$. Dans cette thèse, nous ne considérerons que des CSP discrets (suffisant pour nos problèmes d'ordonnancement) ;
- un ensemble fini de contraintes $\mathcal{C}$. Chaque contrainte $c_i \in \mathcal{C}$ est une relation portant sur un sous-ensemble de variables de $\mathcal{X}$ restreignant les valeurs que ces variables peuvent prendre simultanément.

Instancier une variable consiste à lui affecter une valeur appartenant à son domaine. On note $\mathcal{A} = \{(x_1, v_1), \ldots, (x_i, v_i)\}$ une instantiation des variables $x_1, \ldots, x_i$ où le couple (x_i, v_i) indique que la variable x_i est instantiée à la valeur v_i. On note $var(\mathcal{A})$ l'ensemble des variables instantiées (*i.e.* dont le domaine est réduit à un élément). Une instantiation est dite partielle si $var(\mathcal{A}) \subsetneq \mathcal{X}$ ou totale si $var(\mathcal{A}) = \mathcal{X}$. Une instantiation est dite consistante si elle satisfait simultanément toutes les contraintes associées, *i.e.* qui ne portent que sur des variables instanciées. Une solution d'un CSP est donc une instantiation totale consistante.

Résoudre un CSP consiste à exhiber une unique solution ou à montrer qu'aucune solution existe (le problème est irréalisable). On peut également être intéressé par la détermination de l'ensemble des solutions d'un problème. Un exemple de CSP est défini dans la Figure 4.1 où l'on observe que l'instantiation partielle $\{(x_1, 2)\}$ n'est pas consistante car elle viole la contrainte c_1. En contrepartie, les instantiations $\{(x_1, 0), (x_2, 2), (x_3, 1)\}$, $\{(x_1, 0), (x_2, 2), (x_3, 2)\}$ et $\{(x_1, 1), (x_2, 2), (x_3, 2)\}$ sont totales, consistantes et représentent les 3 solutions de ce problème.

$$\begin{array}{ll} \mathcal{X} & = \{x_1, x_2, x_3\} \\ \mathcal{D}(x_i) & = [0, 2], \forall x_i \in \mathcal{X} \\ \mathcal{C} & = \left\{ \begin{array}{ll} c_1 : & x_1 < x_2 \\ c_2 : & x_1 + x_2 \geq 2 \\ c_3 : & x_1 < x_3 \end{array} \right. \end{array}$$

FIGURE 4.1 – Exemple de CSP

4.1.2 Les problèmes d'optimisation sous contraintes

Certains problèmes de satisfaction de contraintes ont un grand nombre de solutions, certaines étant préférables à d'autres. Cette notion de préférence est généralement modélisée au moyen d'une fonction qui à toute instantiation associe une valeur de coût. Une solution préférée est alors une instantiation qui minimise (ou maximise) cette fonction.

Un problème d'optimisation sous contraintes (CSOP) est un CSP associé à une fonction objectif f. Cette fonction est souvent modélisée par une variable dont le domaine est défini par les bornes supérieures et inférieures de f.

Si nous définissons un CSOP d'après le CSP de la Figure 4.1 et la fonction objectif définie par $min(x_1)$ alors la solution unique $\{(x_1, 1) ; (x_2, 2) ; (x_3, 2)\}$.

4.2 Méthodes de résolution d'un CSP

Les méthodes de résolution de CSP sont par principe génériques, elles ne doivent pas dépendre du problème à résoudre. L'intuition consiste à explorer de manière implicite l'ensemble des instantiations possibles, en distinguant les instantiations non-consistantes et consistantes, jusqu'à trouver les solutions du problème ou prouver qu'il n'existe pas de solutions. Nous décrivons dans cette section les principes de base que sont la construction dynamique de l'arbre de recherche et l'utilisation de la propagation et du filtrage pour élaguer au maximum l'arbre. Nous discutons également de l'intérêt des contraintes globales et de l'utilisation d'heuristiques de branchement. Finalement, nous décrivons la méthode couramment utilisée pour résoudre des problèmes d'optimisation.

4.2.1 Construction d'un arbre de recherche

Résoudre un CSP de manière explicite consiste à générer toutes les instantiations totales possibles puis de vérifier la consistance de chacune. Dans la pratique, cette approche n'est évidemment pas envisageable car le nombre d'instantiations est exponentiel. En effet pour un problème composé de n variables, chacune ayant un domaine de k valeurs, alors il existe k^n instantiations totales. Dans le CSP décrit dans la Figure 4.1, il existe 27 instantiations totales, 3 seulement sont des solutions.

L'algorithme de base utilisé pour résoudre un CSP est l'algorithme de « simple retour arrière » (*backtrack*) [RvBW06]. Partant de l'instantiation vide, il l'étend progressivement en choisissant et en instantiant une nouvelle variable à chaque étape. L'algorithme vérifie alors que l'instantiation partielle courante est consistante avec les contraintes du problème. Dans le cas contraire, la dernière instantiation faite est supprimée et une nouvelle instantiation est alors effectuée. De cette manière, l'algorithme construit un arbre de recherche dont les nœuds représentent les instantiations partielles testées. Cet algorithme permet de tester l'ensemble des instantiations possibles de manière implicite, c'est à dire sans toutes les générer.

La Figure 4.2 représente l'arbre de recherche associé au CSP de la Figure 4.1 construit par l'algorithme de simple retour arrière. Durant la construction de l'arbre de recherche seule 18 instantiations partielles sont testées (contre 27 instantiations totales pour une énumération explicite). L'algorithme de résolution a déduit par exemple qu'en choisissant la valeur 2 pour x_1, toutes les instantiations possibles de x_2 sont inconsistantes car elles violent la contrainte c_1. Il en déduit qu'il n'est alors pas nécessaire de tester les instantiations possibles pour x_3.

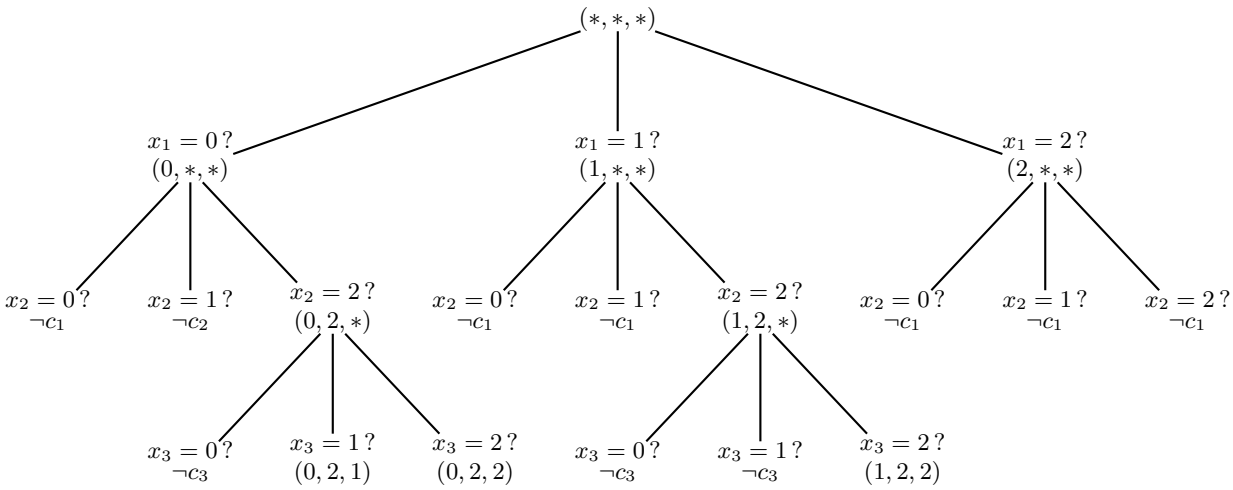


FIGURE 4.2 – Arbre de recherche pour le CSP décrit dans la Figure 4.1. Les triplet (v_1, v_2, v_3) désignent une instantiation respectivement pour les variables x_1, x_2 et x_3.

4.2.2 Filtrage et propagation des contraintes

Le filtrage et la propagation des contraintes [RvBW06] permettent d'améliorer les capacités de résolution des CSP en complément de la construction d'un arbre de recherche. Par exemple, la seule analyse de la contrainte c_1 ($x_1 \leq x_2$) et des domaines de x_1 et x_2 permet de déduire que $(x_1, 2)$ est une instantiation inconsistante. La valeur 2 peut donc être supprimée (ou filtrée) du domaine de x_1 dans notre CSP référence (Figure 4.1). De cette façon, l'arbre de recherche sera réduit car la branche droite n'est plus à explorer.

Plus généralement, à chaque nœud de l'arbre de recherche, c'est à dire à chaque instantiation d'une valeur à une variable l'algorithme place un événement dans une file qui va être analysée par toute les contraintes partageant cette variable. Par un système de déduction locale à chaque contrainte, des valeurs des autres variables de la contrainte peuvent être prouvées comme inconsistantes. La suppression de ces valeurs des domaines est appelée filtrage. À leur tour, les modifications des domaines peuvent inférer de nouvelles inconsistances. Ce principe, appelé propagation, continue jusqu'à ce que la file d'événements soit vide. On atteint alors un point fixe, l'algorithme fait un nouveau choix de variable et de valeur et teste la nouvelle instantiation. Si durant le filtrage d'une contrainte, le domaine d'une variable devient vide, alors l'instantiation courante est inconsistante. Le nœud est fermé et l'algorithme passe au nœud suivant s'il existe. Cette opération permet d'agir sur les domaines de toutes les variables d'un CSP durant la construction de l'arbre de recherche et réduit ainsi le nombre de nœuds composant l'arbre.

En reprenant l'arbre de recherche de la Figure 4.1, la propagation et le filtrage permet d'élaguer fortement l'arbre. Dès la racine, la propagation des contraintes réduit le domaine de x_1 à $[0, 1]$ et les domaines de x_2 et x_3 à $[1, 2]$. L'arbre de recherche complet sera composé de seulement 6 nœuds.

4.2.3 Contraintes globales

Une contrainte globale est définie par un prédicat et porte sur un nombre indéfini de variables. La contrainte globale $allDifferent\,(x_1, \ldots, x_i)$ par exemple, spécifie que toutes ses variables ont des valeurs différentes. Par principe, ces contraintes disposent d'un degré de réutilisabilité supérieure comparé aux contraintes basiques. En effet, ces contraintes sont plus faciles à exprimer et leur écriture est plus concise. Par exemple, l'expression logique (4.1) définie une conjonction de 3 contraintes de différences, équivalente à une contrainte *allDifferent* portant sur 3 variables. Certaines règles ne peuvent également pas s'exprimer autrement que par le biais de contraintes globales. La contrainte de « sac à dos » (*knapsack*) [MT90]

par exemple, qui définit une valeur maximale pour la somme pondérée d'un nombre indéfini de variables (voir équation 4.2).

$$x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3 \equiv \text{allDifferent}\,(x_1, x_2, x_3) \tag{4.1}$$

$$\text{knapsack}\,(x_0, x_1, \ldots, x_i) : x_1 + x_2 + \cdots + x_i \leq x_0 \tag{4.2}$$

Les contraintes globales offrent un algorithme de filtrage dédié réalisant généralement plus de déductions qu'une simple conjonction de contraintes. Par exemple, l'algorithme usuel associé à la contrainte $allDifferent\,(x_1, x_2, x_3, x_4)$ repose sur un algorithme de graphe. Avec $x_1, x_2 \in [1, 4]$ et $x_3, x_4 \in [3, 4]$, il permet de déduire que x_1 et x_2 ne peuvent être instanciées à 3 et 4, ce que ne peut pas réaliser la propagation sur la simple conjonction de contraintes de différence entre ces 4 variables.

4.2.4 Des heuristiques pour guider la recherche

À chaque création d'un nœud de l'arbre de recherche, l'algorithme réalise un point de choix, c'est à dire qu'il va choisir une variable à instancier à une valeur. Le critère de choix de ces deux éléments a évidemment un impact sur la résolution d'un CSP. Imaginons un oracle assurant que l'instantiation de la variable choisie avec la valeur choisie soit toujours consistante, alors l'algorithme trouvera la première solution sans jamais remonter dans l'arbre. Dans la pratique, le degré de complexité des problèmes fait que le développement de telles heuristiques n'est pas possible, il est cependant important de guider au mieux la recherche afin d'améliorer le processus de résolution.

Une approche courante concernant l'heuristique de choix de variable consiste à choisir en priorité les variables les plus dures à instancier, par exemple on sélectionne les variables dont la taille du domaine est minimale. On nomme cette approche « échec d'abord » (*first fail*) [HE80] car elle tente de détecter les instantiations inconsistantes le plus tôt possible dans l'arbre afin de l'élaguer. Cette approche est utilisée naturellement lorsque nous résolvons des problèmes d'ordonnancement par exemple, en plaçant les tâches critiques en priorité.

L'heuristique de choix de valeur joue également un rôle primordial dans la résolution d'un CSP. Elle est par contre souvent plus dépendante du type de résolution du CSP : elle joue un rôle majeur lorsque l'on souhaite calculer une unique solution, alors que son intérêt est plus limité lorsque l'on souhaite obtenir toutes les solutions d'un problème. Une première approche consiste à sélectionner des valeurs présumées consistantes. Une seconde approche, semblable à « échec d'abord » , consiste à sélectionner la valeur la plus critique où les probabilités d'instantiations inconsistantes sont les plus élevées. Si l'heuristique de choix de variable peut être indépendante du CSP, l'heuristique de choix de valeur au contraire est très dépendante du type du problème. Il est en effet nécessaire de comprendre la sémantique de chaque valeur afin de choisir les valeurs critiques ou prometteuses.

Dans l'arbre de recherche présenté dans la Figure 4.1, les heuristiques de choix consistent à traiter les variables dans l'ordre numérique croissant puis à sélectionner les valeurs dans l'ordre croissant. Avec ces deux heuristiques, la première solution est calculée en parcourant 3 nœuds. En sélectionnant les valeurs par ordre décroissant, il est possible de calculer la première solution en parcourant seulement 2 nœuds.

4.2.5 Résolution de problèmes d'optimisation

En pratique, résoudre un problème d'optimisation sous contraintes (CSOP) consiste à trouver une où les solutions du CSP associé dont la valeur de la variable objectif est minimale ou maximale. Dans la pratique, à chaque fois qu'une solution est calculée, une contrainte est ajoutée au CSP et définit les nouvelles bornes de la variable objectif. Dans la plupart des cas, pour résoudre efficacement un CSOP, il est nécessaire de définir précisément les bornes du domaine de la variable objectif : la borne inférieure (respectivement supérieure) si l'on souhaite minimiser (respectivement maximiser) la valeur de la variable.

4.3 Les solveurs de contraintes

Les algorithmes de résolution de problèmes sous contraintes sont implémentés dans des solveurs de contraintes (GeCODE [GEC06], Ilog [ILO], Choco [CJLR06], ...). L'utilisateur décrit son problème grâce

à une API ou un langage dédié mis à disposition par le solveur qui résoudra ensuite le problème. Le Listing 4.1 par exemple, montre la déclaration et la résolution du CSP décrit dans la Figure 4.1 avec l'API du solveur Choco. Ce solveur met à disposition une bibliothèque JAVA permettant la résolution de problèmes de satisfactions de contraintes de d'optimisation. Nous déclarons un nouveau problème initialement vide à la ligne 1 puis nous déclarons les variables du problèmes (ligne 2 à 4) sous la forme de variables entières bornées. De la ligne 6 à 8, nous définissons nos contraintes puis nous les ajoutons au problème (ligne 10 à 12). Finalement, nous demandons au solveur de calculer toute les solutions à notre problème à la ligne 13.

Les solveurs de contraintes mettent à disposition une bibliothèque de contraintes pré-implémentées dans le langage d'exécution du solveur ainsi que différents mécanismes permettant de paramétrer le solveur, par exemple en déclarant ses propres heuristiques de branchement. Le catalogue de contraintes globales [BCR05] définit la sémantique et les propriétés de contraintes standard ainsi que des liens vers leur implémentation pour les solveurs CHOCO ou GeCODE. L'objectif à terme de cet effort de standardisation est de permettre de déclarer des problèmes sans tenir compte des spécificités des solveurs et sans dépendance forte avec ceux-ci.

```
Problem pb = new Problem();
IntDomainVar x1 = pb.makeBoundIntVar("x1", 0, 2);
IntDomainVar x2 = pb.makeBoundIntVar("x2", 0, 2);
IntDomainVar x3 = pb.makeBoundIntVar("x3", 0, 2);

Constraint c1 = pb.lt(x1, x2);
Constraint c2 = pb.geq(pb.plus(x1, x2), 2);
Constraint c3 = pb.lt(x1, x3);

pb.post(c1);
pb.post(c2);
pb.post(c3);
pb.solveAll();
```

Listing 4.1 – Code Source JAVA utilisant l'API Choco pour calculer toutes les solutions du CSP décrit dans l'équation (4.1)

4.4 Conclusion

Nous avons décrit dans ce chapitre les principes de la Programmation Par Contraintes (PPC). Cette approche déclarative permet de modéliser et de résoudre des problèmes combinatoires variés complexes et facilement composables tel que les problèmes d'ordonnancement de placement qui doivent résoudre les gestionnaires de ressources. Les utilisateurs décrivent leurs problèmes en spécifiant des contraintes sur les valeurs que peuvent prendre les différentes variables composant le problème. Un solveur de contraintes est alors chargé de calculer les différentes solutions de ce problème, c'est à dire les différentes instantiations des variables à des valeurs satisfaisant simultanément toutes les contraintes. Cette approche permet une modélisation d'un problème plus simplement qu'avec une approche de type Programmation Linéaire, ne pouvant résoudre que des problèmes définis sous la forme d'équations linéaires ou des problèmes SAT nécessitant une modélisation sous la forme d'équations booléennes.

Le processus de résolution exact de problèmes de contraintes permet de générer efficacement les solutions d'un problème en considérant à la fois le filtrage et la propagation, afin de réduire le temps de résolution des problèmes et des heuristiques de recherche, fournit par l'utilisateur, afin de guider la recherche. Finalement, les efforts récents sur la standardisation de la sémantique des contraintes globales annonce une volonté de s'abstraire des spécificités des bibliothèques de contraintes des solveurs afin de proposer un environnement totalement indépendant de toute implémentation, facilitant la portabilité des contraintes.

L'approche PPC est cependant soumise à plusieurs limitations. Premièrement, la PPC est dédiée à la résolution exacte de problèmes combinatoires. Ainsi le calcul de solutions de problèmes d'optimisa-

tion ou même de satisfaction de contraintes peut nécessiter un temps conséquent limitant le passage à l'échelle du processus de résolution, il importe alors de modéliser le problème de la façon la plus efficace possible. Cette nécessité d'optimiser la modélisation peut alors limiter les possibilités de composition de contraintes. En effet, celles-ci peuvent nécessiter une représentation précise du problème qui ne sera pas nécessairement compatible avec la représentation choisie et il importe alors de réaliser des « ponts » entre les différentes représentations qui peuvent complexifier la modélisation du problème. Finalement, si la théorie de la PPC met en avant une approche déclarative, cet objectif n'est pas encore pleinement atteint. L'effort d'uniformisation des contraintes globales mais également des langages de description de CSP ne permettent pas encore de déclarer un problème sans aucune considération du solveur sous-jacent. Simultanément, il est encore nécessaire de développer dans certaines situations ses propres contraintes globales permettant une résolution efficace d'un problème précis.

Deuxième partie

Contribution

Chapitre 5

Gestion dynamique et autonome de machines virtuelles dans les grappes

Où nous présentons la problématique et la contribution de cette thèse en discutant du manque de dynamicité des gestionnaires de ressources actuels dont l'environnement des grappes ne permet pas d'implémenter de façon fiable les mécanismes nécessaires à la gestion dynamique des tâches. Celle-ci requiert en effet une architecture permettant leur manipulation d'une manière efficace, une approche flexible vis à vis de la variété des critères d'ordonnancement et un système autonome capable de revoir l'agencement des tâches efficacement et en continu afin d'optimiser l'utilisation des ressources. Dans ce contexte, notre contribution consiste en la définition et l'implémentation d'Entropy, un gestionnaire de ressources flexible et autonome à base de machines virtuelles permettant d'implémenter des stratégies d'ordonnancement et de placement complexes par une composition de contraintes.

Sommaire

Les grappes de serveurs sont utilisées principalement par l'intermédiaire de gestionnaires de ressources ordonnançant l'exécution de tâches ayant des besoins en ressources connus. Un des objectifs des gestionnaires de ressources est alors d'utiliser au mieux la capacité de calcul de la grappe pour exécuter les tâches le plus rapidement possible. Il existe différents types de tâches et différentes stratégies d'ordonnancement sélectionnant les tâches à exécuter. On observe cependant que même si les approches purement statiques allouant une quantité de ressources constante pour la totalité de la durée de la tâche ne permettent pas d'occuper au mieux les ressources, celles-ci sont les plus utilisées. En effet, les architectures actuelles des grappes ne supportent généralement pas une gestion dynamique des tâches.

Dans cette thèse, nous abordons la problématique de la gestion dynamique des tâches en définissant et en implémentant une solution autonome et flexible utilisant une architecture à base de machines virtuelles et une approche pour leur placement sur les nœuds de calcul reposant sur la programmation par contraintes. Dans ce chapitre, nous définissons d'abord le principe de gestion dynamique des tâches en décrivant des critères permettant une utilisation efficace des ressources et les mécanismes associés.

Nous réalisons ensuite un parallèle entre ces critères et le fonctionnement des différents gestionnaires de ressources utilisés en production. Dans une troisième section, nous décrivons les pré-requis pour une gestion dynamique des tâches. Nous terminons ce chapitre par la description de notre approche.

5.1 La gestion dynamique des tâches dans les grappes

En se basant sur l'étude des différents modèles de tâches et les différents algorithmes d'ordonnancement décrit dans le chapitre 2, il est possible de dégager différents critères permettant d'optimiser le taux d'occupation des ressources. Feitelson *et al.* [FRS+97] proposent cinq recommandations portant sur le modèle de tâches et la stratégie d'ordonnancement :

1. **Utiliser des tâches au moins malléables**
   Les tâches malléables et évolutives supportent une allocation dynamique des ressources si elles sont couplées à un partitionnement adaptatif ou dynamique. Si une application a des besoins en ressources variables au cours du temps (si elle dispose de plusieurs phases par exemple), alors une allocation dynamique des ressources permet de suivre les besoins de l'application et supprime la tendance à la sur-réservation de ressources, observable lors de l'utilisation d'une partition variable.

2. **Considérer les caractéristiques des tâches**
   Les tâches variables laissent une grande responsabilité à l'utilisateur qui doit déclarer les besoins en ressources et le temps nécessaire à l'exécution de celles-ci. L'estimation du temps est souvent imprécise et surestimée afin d'éviter tout risque de suppression de la tâche durant son exécution [Wei98]. L'utilisateur peut également demander une quantité de ressources supérieure à ses besoins afin de prévoir une éventuelle diminution de la partition accordée (par exemple si un nœud tombe en panne entre la soumission et le démarrage de la tâche). Même si les utilisateurs restent peu précis dans la déclaration de leurs tâches, ces informations permettent de définir une borne supérieure concernant les besoins en ressources. Il est alors possible d'exécuter des tâches des caractéristiques variées.

3. **Baser l'allocation des ressources sur la charge courante du système**
   Dans le cadre de tâches malléables ou évolutives, il est important de tenir compte de l'état courant de la grappe afin d'adapter la taille des partitions à la charge courante de celle-ci. Dans cette situation, le gestionnaire de ressources doit connaître aussi précisément que possible le taux d'utilisation des ressources des différents composants des tâches afin de compléter les informations fournies par l'utilisateur. Ce croisement d'informations permet ainsi d'employer une quantité de ressources plus en adéquation avec les besoins réels des applications en fonctionnement.

4. **Préférer un ordre de file non strict et un partitionnement temporel**
   La gestion stricte des files d'attente, comme c'est le cas avec l'algorithme FCFS, réduit le taux d'occupation des ressources en empêchant les petites tâches de se lancer rapidement. Si les utilisateurs souhaitent une exécution au plus tôt, il est alors nécessaire de proposer un ordre de file non strict en utilisant un algorithme de *backfilling* tel que EASY ou même *First Fit*. De plus, l'exécution de la tâche sans préemption, comme lors d'un partitionnement spatial, réduit les possibilités d'occupation des ressources. Le partitionnement temporel au contraire accélère le traitement des tâches en les exécutant même partiellement, dès que suffisamment de ressources sont disponibles.

5. **Fournir un support pour la migration**
   La migration permet d'obtenir un meilleur taux d'occupation des ressources en limitant la fragmentation des ressources dans les grappes par une réaffectation des ressources à la volée. Cette opération est facultative dans un environnement utilisant des partitions variables, elle est par contre indispensable dans un environnement à base de partitions dynamiques afin de s'adapter facilement à l'augmentation et la diminution de la taille de celles-ci. Si la préemption locale est suffisante et permet d'utiliser des modèles de partionnement temporel implicite et explicite, la possibilité de migrer les processus sur différents nœuds apporte une flexibilité supérieure. Il faut cependant considérer le temps de migration du processus et de la gestion de ses dépendances notamment pour les environnements où les nœuds ne partagent pas leurs mémoires vives.

Les différents critères décrit par Feitelson *et al.* prônent le besoin de davantage de dynamicité et de flexibilité dans les grappes. Nous définissons la gestion dynamique des tâches comme un ensemble de mécanismes permettant de manipuler l'état et le placement des composants d'une tâche à la volée. Dans la pratique, cela consiste à pouvoir suspendre une tâche en cours d'exécution si nécessaire afin de pouvoir la relancer ultérieurement mais également à pouvoir migrer ses différents composants à la volée durant son exécution et si possible sans arrêt de service. Ces actions ayant un impact sur la disponibilité des ressources dans la grappe, cela implique également de pouvoir allouer ou désallouer des ressources aux différents composants lorsque le besoin se présente. La gestion dynamique des tâches permet ainsi de développer des stratégies de placement et d'ordonnancement avancées, considérant les critères d'efficacité évoqués ci-dessus et permettant une meilleure utilisation des ressources de la grappe.

5.2 L'ordonnancement des tâches dans la pratique

L'analyse des principaux gestionnaires de ressources gratuits comme OAR [CDCG+05], Sun Grid Engine [Gen01], MAUI [JSC01] ou commerciaux [ET05], montre que ceux-ci utilisent un modèle de tâches rigide couplé à un partitionnement spatial variable. La stratégie d'ordonnancement combine le plus souvent des files d'attente de type FCFS avec *EASY-Backfilling* et différentes classes de tâches, avec des niveaux de priorités différents. Cette observation est confirmée en analysant les différentes traces d'activité des grappes présentées sur le site *Parallel Workload Archive* [PWA]. Ce site regroupe les traces de l'activité de 25 grappes ainsi qu'une description de leur fonctionnement. Sur ces 25 traces, la politique d'ordonnancement est précisée pour 24 d'entre elles et on observe alors que seulement deux grappes utilisent une stratégie de type *gang scheduling* avec uniquement de la préemption locale. Les 22 autres approches proposent un partitionnement uniquement spatial avec des algorithmes d'ordonnancement de type FCFS avec ou sans *backfilling*, ou équitable avec des priorités différentes entre chaque file d'attente. Encore une fois, les gestionnaires de ressources utilisent des tâches rigides avec un partitionnement des ressources spatial, fixe ou variable.

Nous considérons que ces approches limitent les possibilités d'utilisation des ressources des grappes. En effet, le modèle de tâche rigide avec partitionnement variable est adapté aux tâches ayant une consommation constante en ressources. Si les applications ont des besoins en ressources variables dans le temps, il n'est pas possible d'adapter la partition de ressources qui a été attribuée à la tâche. La sur-réservation des ressources permet d'assurer une exécution optimale de l'application mais produit un décalage entre le taux de réservation des ressources et le taux d'utilisation de celles-ci. L'utilisation d'un partitionnement spatial avec une exécution des tâches sans préemption limite également les possibilités d'ordonnancement en rendant difficile l'exécution des tâches requérant une grande quantité de ressources ou un grand temps de calcul.

Nous pensons que l'environnement logiciel des grappes actuelles n'est pas adapté à une gestion dynamique des tâches. Dans la pratique, chaque composant d'une tâche est isolé dans un processus qui est exécuté sur un nœud de calcul exécutant un système d'exploitation standard. Des outils de capture d'état ou de migration de processus ont été développés afin d'améliorer la flexibilité de l'ordonnancement des tâches, de réaliser de l'équilibrage de charges ou de la reprise en cas de faute. Ces outils peuvent cependant nécessiter un environnement particulier, incompatible avec l'application à exécuter ou un effort de ré-ingéniérie pour adapter l'application. D'autre solutions tel que BLCR [HD06b] ou le processus de migration du système CONDOR [LTBL97, TTL05] ont été développées proposant des mécanismes plus efficaces, plus rapides et moins invasifs. L'utilisation de telles bibliothèque ne nécessite pas de recompiler l'application. Une application dispose cependant de dépendances avec des librairies, des descripteurs de fichiers ouverts ou des connexions réseaux qui ont du mal à être pris en compte. La détection et la gestion de ces dépendances résiduelles a limité le déploiement de ces mécanismes [MDP+00] pourtant nécessaire pour une gestion dynamique des tâches.

La gestion dynamique des ressources implique des modèles de tâches malléables ou évolutifs et des partitions adaptatives ou dynamiques. Cela nécessite à la fois un environnement d'exécution permettant un contrôle des ressources attribuées aux processus, mais également que les applications des utilisateurs puissent prendre en compte l'arrivée ou le départ de ces ressources. Dans la pratique, cette coopération entre l'application et le gestionnaire de ressources nécessite un intergiciel ou un mode de développement précis [UCL04] qui limitent l'utilisabilité de tels modèles de tâches : d'une part les utilisateurs n'ont pas

toujours comme préoccupation de développer leurs applications pour un environnement d'exécution précis et d'autre part, les administrateurs veulent limiter la spécialisation de la grappe, si celle-ci est destinée à exécuter des applications ayant des profils variés.

5.3 Expression des besoins pour une gestion dynamique des tâches

L'observation des critères définissant une gestion dynamique des tâches et l'analyse des différents gestionnaires de ressources actuels permettent de dégager différents pré-requis nécessaires au développement d'un tel environnement. Nous décrivons dans cette section ces différents points.

5.3.1 Un support adapté à la manipulation des tâches

Le développement d'algorithmes d'ordonnancement avancés nécessite de pouvoir migrer ou de capturer l'état des différents composants d'une tâche. Cet environnement doit permettre une isolation forte entre les différentes tâches, à la fois pour une question de protection en cas d'erreur dans une application, mais également pour un meilleur contrôle des ressources. Simultanément, l'environnement doit fournir un moyen d'encapsuler les composants dans des blocs facilement manipulables.

Finalement, ce support doit permettre de limiter les efforts d'adaptation des applications à exécuter sur la grappe. Les applications peuvent en effet avoir besoin d'environnements d'exécution spécifiques comprenant un système d'exploitation et un intergiciel précis. Toutes ces dépendances doivent pouvoir être incluses dans un bloc primitif. L'application dispose alors de son environnement d'exécution d'origine, manipulable de manière non-invasive et transparente.

5.3.2 Une approche flexible

Sélectionner les tâches à exécuter à un moment précis et donc manipuler leur état s'apparente à des problèmes d'ordonnancement en-ligne tandis que placer les différents composants des tâches en cours d'exécution sur des nœuds s'apparente à un problème de placement. Ces deux classes de problèmes appartiennent à la classe des problèmes combinatoires. Les gestionnaires de ressources sont paramétrables par les administrateurs afin de s'adapter aux mieux aux spécificités de la grappe. Pour chaque grappe, l'administrateur spécifie en effet des règles liées à l'ordonnancement des tâches mais également au placement de leurs composants.

La spécialisation de l'ordonnancement peut porter sur les caractéristiques des tâches ou sur des critères politiques. L'administrateur peut par exemple demander à utiliser un algorithme d'ordonnancement sélectionnant les tâches à exécuter en fonction de leur date d'arrivée et de leur besoin courant en ressources tout en traitant en priorité les tâches soumises par un groupe d'utilisateurs précis.

La spécialisation du placement des composants des tâches consiste à sélectionner les nœuds accueillant les composants des tâches en cours d'exécution selon différents critères. L'administrateur peut par exemple demander à placer tous les composants sur un nombre de nœuds minimum afin de réduire la consommation énergétique en éteignant les nœuds inutilisés. Il peut également préciser plus finement ses besoins en demandant à ce que les composants de certaines tâches soit nécessairement exécutés sur des nœuds différents de manière à augmenter la fiabilité de l'exécution de la tâche.

Les combinaisons des différents paramètres pour spécialiser le gestionnaire de ressources sont potentiellement infinies, Il n'est donc pas pensable de développer une architecture prévoyant toute ces combinaisons. Une solution consiste alors à pouvoir composer soi-même sa stratégie de gestion des tâches en déclarant les différents critères à considérer. L'ordonnancement et le placement de tâches sont des problèmes combinatoires complexes, il importe donc de choisir une approche permettant la résolution de problèmes combinatoires de manière efficace mais aussi flexible.

Les approches heuristiques proposent des algorithmes résolvant des problèmes d'optimisation combinatoires. Ces algorithmes ne sont pas nécessairement exacts, c'est à dire qu'ils ne produisent pas nécessairement une solution optimale, cette carence étant théoriquement compensée par leur rapidité de calcul. Surtout, une heuristique est un algorithme ad-hoc à un problème précis basé sur une modélisation

spécifique du problème. De ce fait, la composition d'heuristiques pour l'application de plusieurs critères simultanément, n'est pas généralement possible et surtout n'assure pas une solution cohérente. Le développement d'une heuristique globale n'est également pas envisageable d'une part à cause de la complexité d'un tel développement, d'autre part car l'ensemble des critères applicables n'est pas connu a priori.

Les systèmes à base de règles [HR85] permettent d'exécuter une action lorsque des faits valident certaines conditions. Une règle est constituée d'un antécédent déclarant une suite de conditions et d'un conséquent spécifiant les actions à exécuter. Les différentes règles sont insérées dans un moteur d'inférence chargé de déterminer les règles à appliquer d'après l'observation de faits. Cette approche permet un certain niveau de flexibilité en ajoutant ou en supprimant des règles en fonction des besoins. De plus, l'écriture des règles correspond à un mode de raisonnement usuel où l'on considère des actions à réaliser en fonction de situations particulières. Cette approche est cependant soumise à deux problèmes majeurs. Premièrement, il est difficile de réaliser une amélioration globale de la gestion des ressources si les antécédents portent sur une vue locale de l'environnement. Deuxièmement, la sélection et l'exécution des règles est soumise à des problèmes de conflits et d'effets de bord : le résultat de l'application d'un ensemble de règles n'est pas déterministe et peut être incohérent si l'ordre d'analyse des règles a un impact sur le processus de sélection ou lorsque plusieurs règles portent sur une même condition ou exécutent des actions sur les mêmes entités.

La Programmation Linéaire en Nombre Entier [NW88] (PLNE) résout de manière exacte et générique des problèmes combinatoires. Dans cette approche, les utilisateurs décrivent un modèle composé d'équations linéaires, un solveur calcule alors les solutions satisfaisant toutes les équations simultanément. La PLNE est une solution performante capable de résoudre des problèmes complexes cependant, la linéarisation des critères soumises au solveur n'est pas naturelle est peut s'avérer complexe pour un non-expert. De plus, un processus de résolution efficace peut impliquer un modèle peu explicite limitant les possibilités d'y ajouter ses propres contraintes de façon fiable.

La Programmation Par Contraintes (PPC) présentée dans le chapitre 4 est également une approche de résolution exacte et générique de problèmes combinatoires. La modélisation d'un problème à base de contraintes consiste à définir des variables, chacune prenant sa valeur dans un domaine précis et des relations (les contraintes) interdisant l'instantiation simultanée de certaines variables à des valeurs précises. Cette approche permet de modéliser des problèmes combinatoires très variés de plus, le paradigme déclaratif de cette approche permet de définir d'une façon naturelle les critères que l'on souhaite voir satisfaits dans chaque solution d'un problème. À partir d'une modélisation de base, l'utilisateur peut ajouter au besoin différentes contraintes et définir son problème spécifique sans possibilité d'indéterminisme dans le processus de résolution.

5.3.3 Un système autonome

Dans le contexte de la gestion dynamique de tâches, les critères favorisant un fort taux d'occupation des ressources comme le *gang scheduling* avec concentration des tâches mettent en avant le besoin d'une architecture adaptant en temps réel la liste des tâches en cours d'exécution ainsi que leur placement sur les nœuds de calcul. Un environnement de gestion dynamique de tâches doit donc pouvoir observer l'état courant de la grappe ainsi que les besoins courants en ressources des différentes composants des tâches et proposer un nouvel agencement des tâches améliorant le taux d'occupation des ressources. Cette auto-adaptation est guidée par les différents critères paramétrant l'ordonnanceur de la grappe.

En 2003, Kephart *et al.* [KC03] proposent le concept d'informatique autonome (*autonomic computing*). Cette approche suggère que les systèmes informatiques s'adaptent eux même à leur environnement plutôt que de réaliser cette adaptation manuellement par un utilisateur qualifié. Quatre catégories d'auto-adaptation ont été mises en avant :

- **l'auto-réparation :** le système détecte, diagnostique et répare de lui même des erreurs logicielles et matérielles ;
- **l'auto-protection :** le système se défend automatiquement contre des attaques ou des erreurs en cascade. Il peut également anticiper ces événements par le biais d'une analyse des messages d'avertissements ;

- **l'auto-configuration :** le système se configure automatiquement d'après des règles de haut niveau spécifiées par un utilisateur sans connaissances précises des mécanismes interne de l'application ;
- **l'auto-optimisation :** le système cherche continuellement les différentes opportunités d'améliorer ses performances et son efficacité.

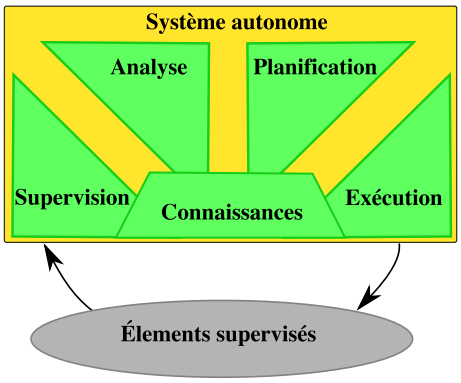


FIGURE 5.1 – Boucle de contrôle d'un système autonome

L'informatique autonome se focalise sur la notion de boucle de contrôle pour réaliser son auto-adaptation (voir Figure 5.1). Cette boucle peut être passive, et dans ce cas elle est lancée manuellement ou périodiquement. L'auto-adaptation peut également être active et répondre à différents événements extérieurs. Le processus d'auto-adaptation passe d'abord par une phase de supervision. Durant celle-ci, le système observe les différents composants supervisés afin d'obtenir des informations qui seront utilisées durant la phase d'analyse où le système statue sur l'intérêt d'une adaptation et décrit les actions à réaliser. La phase de planification organise alors les différentes actions à réaliser et la phase d'exécution lance le processus en exécutant les différentes actions. Durant tout le processus d'auto-adaptation, une base de connaissances permet de renseigner le système sur les différentes adaptations envisageables et fournit également des méthodes de résolution permettant de détecter les besoins d'adaptation.

5.4 Proposition : un gestionnaire de ressources autonome à base de machines virtuelles

Nous proposons dans cette thèse un gestionnaire de ressources basé sur un environnement à base de machines virtuelles fournissant les mécanismes nécessaires à la gestion dynamiques des tâches. Ce système, baptisé « Entropy » est un gestionnaire de ressources autonome optimisant en continu l'agencement des machines virtuelles. Un module de décision indique l'état que l'administrateur souhaiterait obtenir et un module de planification assurant la transition depuis l'état courant de la grappe. Nous détaillons dans cette section les différentes contributions de cette thèse.

5.4.1 Une grappe de serveurs à base de machines virtuelles

L'analyse des fonctionnalités d'isolation et de manipulation des machines virtuelles discutées dans le chapitre 3 démontre que celles-ci sont des supports valides pour une gestion dynamique des tâches dans les grappes. En embarquant chaque composant des tâches dans leur propre machine virtuelle et en exécutant un hyperviseur sur chaque nœud de calcul, nous manipulons les machines virtuelles comme des blocs primitifs sans se soucier des dépendances résiduelles. L'isolation fournie par l'hyperviseur permet un contrôle fin des ressources utilisées par les machines virtuelles tandis que les fonctionnalités de migration, de suspension ou de reprise d'activité permettent de les manipuler facilement les machines virtuelles et de façon transparente pour l'utilisateur.

À la vue des différents types d'hyperviseurs disponibles, il est important de choisir une solution adaptée aux machines virtuelles afin d'obtenir les meilleurs performances possibles. Ainsi, un hyperviseur

proposant de la virtualisation matérielle, assistée ou non, peut héberger des machines virtuelles exécutant des systèmes d'exploitations très variés (GNU/Linux, Windows, OS X, ...) mais fournit un niveau de performances en deça d'un hyperviseur utilisant de la para-virtualisation.

L'utilisation des outils manipulant les machines virtuelles, notamment la migration, implique certaines contraintes relatives à la gestion des images des disques virtuels. Si des systèmes de fichiers distribués dédiés aux machines virtuelles et utilisant les disques locaux commencent à apparaître [MAC+08, VMF07], la solution utilisée majoritairement dans les grappes de serveurs met à disposition les images disques des machines virtuelles depuis un réseau de stockage SAN pour assurer leur accessibilité depuis tous les nœuds. Pour les opérations de suspension et de reprise d'activité, il est possible d'utiliser les disques des nœuds de calcul, avec éventuellement une copie de l'image de la mémoire lorsque la reprise d'activité de la machine virtuelle se fait sur un nœud différent.

5.4.2 Une approche auto-adaptative

La gestion dynamique des tâches nécessite que l'état des tâches ainsi que le placement de leurs composants soit remis en cause lorsqu'ils ne satisfont plus les critères définis par l'administrateur. Simultanément, l'observation des différentes politiques d'ordonnancement et de placement des tâches dans les grappes montre un besoin de flexibilité dans l'expression de l'ordonnanceur. Dans ce contexte, une approche autonome permet de satisfaire ces besoins en proposant une auto-configuration du module d'analyse calculant un nouvel agencement des tâches et une auto-optimisation en continu afin de satisfaire les critères de satisfaction de l'ordonnanceur. Un module de planification a alors la charge d'assurer la transition entre l'agencement courant et l'agencement souhaité par le module d'analyse.

5.4.2.1 Un module d'analyse composable

La gestion dynamique des tâches doit ordonnancer des tâches, en sélectionnant les tâches à exécuter en fonction de différents critères et de placer de machines virtuelles en choisissant les nœuds qui héberge les machines virtuelles des différentes tâches en cours d'exécution. Nous avons décrit précédemment que les critères affectant ces problèmes combinatoires étaient variés et que l'ensemble des critères à appliquer durant cette phase n'était pas constant. De plus, plusieurs critères peuvent porter sur un même sous-ensemble d'entités (machines virtuelles ou nœuds). Il est donc nécessaire de choisir une approche assurant une modélisation et une résolution de problèmes combinatoires spécifiques par une déclaration naturelle des critères à considérer.

La Programmation Par Contraintes (PPC) présentée dans le chapitre 4 apparaît comme une approche viable pour la définition d'un module d'analyse flexible. En modélisant l'état des machines virtuelles ainsi que la notion de placement de celles-ci sur les nœuds par un problème de satisfaction de contraintes (CSP) de base, nous pouvons composer des problèmes de placement et d'ordonnancement en-ligne spécifiques en ajoutant les différents contraintes que l'administrateur souhaite voir satisfaits. L'approche déclarative de la PPC permet une expression en intention de ces contraintes, le solveur est alors en charge de calculer un nouvel agencement des tâches satisfaisant simultanément toutes les contraintes.

5.4.2.2 La planification de l'adaptation

Durant la phase d'analyse, le système peut proposer un nouvel agencement des machines virtuelles permettant d'obtenir de meilleures performances et de maintenir les contraintes de l'administrateur. Dans cette situation, le module de planification détecte les différentes actions à réaliser afin d'assurer une transition entre l'agencement courant des machines virtuelles et le nouvelle agencement. Cette adaptation peut nécessiter d'exécuter différentes actions sur les machines virtuelles. Dans le cadre de notre thèse, nous considérons les actions de lancement, d'arrêt, de migration, de suspension et de reprise d'activité. L'exécution d'une action nécessite des ressources CPU et mémoire pour les nœuds impliqués durant la totalité de la durée de l'action. Les performances de ces nœuds sont alors temporairement réduites durant toute la durée de l'exécution de l'action. De plus, les actions de migration, de lancement et de reprise d'activité dispose de pré-conditions à leur exécution. Il est donc nécessaire de planifier l'ordre d'exécution de ces actions.

La planification des actions d'adaptation est assimilable à un problème d'ordonnancement. Cela consiste à définir un plan d'exécution des actions assurant la faisabilité de celles-ci. Il est également nécessaire d'optimiser ce plan de façon à réduire son temps d'exécution ainsi que son impact sur les performances de la grappe au minimum. Pour réaliser cette planification, notre solution repose sur une approche mixte où la création du plan d'exécution est réalisée par une heuristique tandis que la réduction du temps d'application et de son impact est réalisée par de la PPC.

5.5 Conclusion

Nous avons discuté dans ce chapitre de la nécessité d'une gestion dynamique des tâches dans les gestionnaires de ressources pour obtenir une utilisation efficace des ressources dans les grappes de serveurs. Cette gestion implique de pouvoir manipuler l'état des différents composants des tâches et leurs positions dans la grappe en utilisant des mécanismes de préemption et de migration. Ces mécanismes représentent la base nécessaire à l'utilisation de stratégies d'ordonnancement avancées permettant un fort taux d'occupation des ressources.

Les gestionnaires de ressources utilisés en production sont généralement basés sur des nœuds de calcul avec un système d'exploitation standard qui exécute les tâches en embarquant leurs différents composants dans des processus. Bien que les mécanismes de migration et de capture d'état existent pour les processus, leur implémentation possède un certains nombre de limitations telles que la gestion des dépendances résiduelles ou le maintien des connexions réseaux qui empêche leur utilisation dans les gestionnaires de ressources standard. Cette architecture, peu adaptée aux besoins de la gestion dynamique des tâches, empêche l'utilisation de stratégies d'ordonnancement avancées.

Une gestion des tâches dynamique nécessite d'abord un environnement supportant les opérations de migration et de préemption des tâches et ce, sans impliquer l'adaptation de l'application. L'observation de gestionnaires de ressources et de différentes stratégies d'ordonnancement et de placement montre également un besoin de flexibilité important permettant à l'administrateur de personnaliser sa stratégie d'ordonnancement en considérant des critères de sélection portant sur les tâches à exécuter ou de satisfaction sur leur placement. Finalement, la gestion dynamique des tâches implique de pouvoir automatiquement adapter leur état et leur placement dans la grappe en fonction de l'état courant de celle-ci afin de maintenir un taux d'utilisation des ressources important.

L'observation des pré-requis pour une gestion dynamique des tâches nous a permis de définir les différents éléments de contribution de cette thèse. Nous proposons un gestionnaire de ressources autonome pour des tâches composées de machines virtuelles. Cette architecture fournit en standard un support non-invasif pour l'exécution des tâches des utilisateurs, limitant les efforts d'adaptations et disposant mécanismes de capture d'état et de migration. Notre gestionnaire de ressources, baptisé Entropy, propose une approche autonome pour la gestion de l'état et du placement des machines virtuelles. Un module d'analyse basé sur de la Programmation Par Contraintes (PPC) permet après l'observation de l'état courant de la grappe de calculer un nouvel état optimisant l'agencement des machines virtuelles. La flexibilité offerte par la PPC permet à l'administrateur de composer son module d'analyse à la volée en indiquant des contraintes qui doivent être satisfaites par le nouvelle état. Un module de planification est ensuite chargé d'assurer la transition entre l'état courant de la grappe et le nouvel agencement des machines virtuelles. Ce processus assure la faisabilité de toute les action composant la transition et cherche à réduire sa durée et son impact sur les performances.

Dans les chapitres suivants, nous détaillons les différents éléments de contribution de cette thèse en discutant dans le chapitre 6 des concepts de base du prototype Entropy puis dans le chapitre 7 nous présentons un module de décision réalisant de la consolidation dynamique de machines virtuelles. Cela consiste à diminuer les coûts de fonctionnement ou d'augmenter la capacité de traitement de la grappe en concentrant les différents machines virtuelles en cours d'exécution sur le minimum de nœuds possible tout en maintenant leurs besoins en ressources. Dans le chapitre 8, nous discutons du processus de reconfiguration assurant la transition entre l'état courant de la grappe et le nouvel état calculé par un module de décision. Le chapitre 9 élargit le champs d'application d'Entropy en définissant la notion de changement de contexte dans les grappes virtualisées pour faciliter le développement d'ordonnanceurs. Finalement, nous discutons de l'évaluation de notre prototype et de ses différents modules dans le chapitre 10.

Chapitre 6

Entropy - Un gestionnaire de placement dynamique autonome

Où nous présentons l'architecture générale du prototype Entropy, un gestionnaire de machines virtuelles proposant une auto-adaptation de l'état et de leur position dans la grappe selon des contraintes définies par les utilisateurs. Nous discutons des interfaces permettant de configuration Entropy pour un environnement de grappe spécifique, par le biais d'adaptateurs pour un système de supervision et de pilotes pour l'exécution d'actions d'adaptation. Finalement, nous décrivons la base de connaissances d'Entropy et son API dédiée à l'écriture de modules de décision reposant sur la programmation par contraintes.

Sommaire

DANS le cadre de cette thèse, nous avons développé le prototype Entropy [ENT]. Entropy est un gestionnaire de ressources autonome pour des tâches composées de machines virtuelles permettant l'auto-adaptation du placement et de l'état de machines virtuelles sur des nœuds de calcul. La résolution des problèmes de placement des machines virtuelles et d'ordonnancement des tâches repose sur la Programmation Par Contraintes (PPC). Cette approche permet de développer et de composer facilement des modules de décisions définissant des stratégies de placement ou d'ordonnancement. Entropy a permis l'implémentation et la validation des contributions exposées dans cette thèse. L'application Entropy ainsi que la totalité de son code source est disponible publiquement [ENT] sous licence LGPL [LGP07] depuis janvier 2009.

Dans ce chapitre, nous discutons de l'architecture générale d'Entropy. Dans une première section, nous décrivons les concepts généraux d'Entropy en définissant un cycle de vie pour les machines virtuelles, la représentation de l'état d'une grappe et la boucle de contrôle permettant l'auto-optimisation du placement des machines virtuelles. Nous discutons ensuite des mécanismes permettant d'interfacer Entropy

avec différents environnements logiciels, puis nous décrivons les interfaces disponibles actuellement. Finalement, nous exposons la base de connaissances d'Entropy qui propose une API reposant sur la PPC pour le développement de stratégies d'ordonnancement et de placement complexes.

6.1 Architecture générale

Dans cette section, nous exposons les concepts permettant la gestion des machines virtuelles. Après une description du cycle de vie d'une machine virtuelle, nous définissons la notion de configuration décrivant de façon abstraite le placement des machines virtuelles dans une grappe ainsi que la distribution des ressources. Finalement nous présentons la boucle de contrôle d'Entropy permettant l'auto-optimisation du placement des machines virtuelles.

6.1.1 Cycle de vie des machines virtuelles

Une tâche est un ensemble de machines virtuelles dédié à l'exécution d'une application parallèle. Celle-ci est soumise par un utilisateur à Entropy. Le cycle de vie d'une tâche correspond aux différents états traversés par les machines virtuelles durant leur prise en compte par Entropy. Nous considérons d'un point de vue théorique que le changement d'état d'un groupe de machines virtuelles appartenant à la même tâche est une opération atomique, nous reviendrons cependant sur cette hypothèse dans le chapitre 9.

La Figure 6.1 décrit le cycle de vie d'une machine virtuelle. Au moment de la soumission d'une tâche, les différentes machines virtuelles la composant sont dans l'état `Attente`. L'action *lancement* démarre une machine virtuelle sur un nœud de calcul et la place dans l'état `Exécution`. Dans cet état, une machine virtuelle peut être mise en pause par l'action *suspension*. L'état de la machine virtuelle est écrit sur un support de stockage, elle n'est plus exécutée par un hyperviseur et passe dans l'état `Endormi`. Une machine virtuelle dans l'état `Endormi` peut être relancée depuis son image sur un nœud de calcul par l'action *reprise*. Il est possible de migrer une machine virtuelle dans l'état `Exécution` sur un autre nœud de calcul par l'action *migration*. Dans cette situation, l'état de la machine virtuelle reste inchangé. Finalement une machine virtuelle dans l'état `Exécution` peut être stoppée par le biais de l'action *arrêt*. Dans ce cas, le nœud exécutant la machine virtuelle la détruit et libère les ressources que celle-ci consommait. Le pseudo-état `Prêt` est composé des états `Attente` et `Endormi`. Ces deux état précèdent l'état `Exécution` et si les actions permettant de passer dans cet état sont différentes, elles sont équivalentes si l'on considère uniquement la disponibilité de la machine virtuelle.

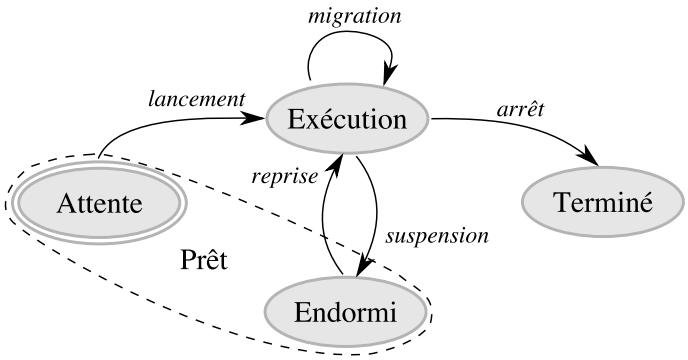


FIGURE 6.1 – Cycle de vie d'une machine virtuelle

6.1.2 Description de l'état d'une grappe

Nous définissons l'état d'une grappe par une configuration, une représentation abstraite décrivant les différents nœuds de calcul et machines virtuelles considérées par Entropy, la distribution des ressources, l'état des machines virtuelles et leurs positions courantes dans la grappe.

Un nœud est caractérisé par un identifiant (son nom d'hôte) et les ressources disponibles pour héberger des machines virtuelles. Une tâche est caractérisée par un identifiant et l'ensemble de ses machines virtuelles. Une machine virtuelle est caractérisée par un identifiant (son nom), son état et ses besoins en ressources.

Les ressources actuellement considérées par Entropy sont les ressources CPU et mémoire des nœuds et des machines virtuelles. La capacité mémoire d'un nœud correspond à la quantité de mémoire vive utilisable par les machines virtuelles dans l'état `Exécution` hébergées par le nœud. La capacité CPU correspond à une valeur abstraite, dépendante du nombre de processeurs et de cœurs disponible sur un nœud ainsi que de leurs fréquences de fonctionnement. Le besoin en mémoire d'une machine virtuelle correspond à la quantité de mémoire nécessaire à son fonctionnement et le besoin en CPU est une valeur indiquant la quantité de ressources permettant d'obtenir des performances optimales. Quand une machine virtuelle est dans l'état `Exécution` ses besoins CPU et mémoire correspondent à ses besoins courant. Quand une machine virtuelle est dans l'état `Attente`, ses besoins en CPU sont considérés comme nuls cependant ces besoins mémoires sont pris en compte. Finalement, quand une machine virtuelle est dans l'état `Endormi` alors ses besoins en ressources correspondent à ses besoins au moment de sa suspension.

Chaque machine virtuelle dans l'état `Exécution` est nécessairement hébergée par un nœud de calcul. Une machine virtuelle dans l'état `Endormi` peut éventuellement utiliser le disque local d'un nœud de calcul pour stocker son image mémoire. Finalement, les machines virtuelles dans l'état `Attente` sont hébergées sur un nœud virtuel qu'on l'on nomme « incubateur ». La Figure 6.2 décrit graphiquement une configuration définissant la capacité en ressource CPU et mémoire de chaque nœud, le placement et les besoins de chaque machine virtuelle dans l'état `Exécution`.

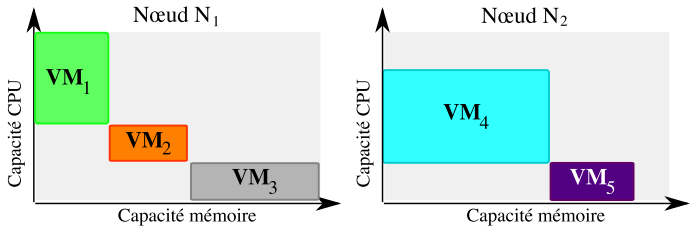


FIGURE 6.2 – Représentation graphique d'une configuration de 2 nœuds hébergeant 5 machines virtuelles en cours d'exécution.

6.1.3 Boucle de contrôle

Entropy réalise une optimisation en continu du placement des machines virtuelles sur les nœuds de la grappe en suivant les principes de l'informatique autonome. Son architecture suit donc la notion de boucle de contrôle infinie introduite dans le chapitre 5. Le déclenchement de la boucle de contrôle est passif. La boucle est relancée périodiquement ; cependant l'adaptation est réalisée uniquement lorsque la phase d'analyse décide qu'une adaptation est nécessaire ou améliore significativement le placement des machines virtuelles.

La Figure 6.3 décrit l'architecture d'Entropy. Le module de supervision est un système de supervision pour grappe de serveurs. Il fournit l'ensemble des informations nécessaire à l'extraction d'une configuration représentant l'état courant de la grappe. Ce module peut être spécifique à une grappe ; de plus il existe plusieurs systèmes de supervision pour grappes utilisés en production (Ganglia [MCC04], NWS [WSH99] ou Nagios [ID07] par exemple). Nous avons donc choisi de fournir aux administrateurs des moyens pour intégrer Entropy à leur environnement logiciel de gestion de grappes plutôt que d'imposer notre propre environnement. Cette intégration passe par le développement d'un adaptateur compatible avec le système de supervision permettant d'extraire une configuration utilisable par le module de décision.

Le module de décision est responsable de l'optimisation de l'ordonnancement des tâches ou du placement des machines virtuelles composant les tâches en cours d'exécution. Dans la pratique, ce module observe la configuration courante et calcule une nouvelle configuration répondant aux besoins de l'administrateur. Une approche reposant sur la Programmation Par Contraintes (PPC) assure la définition et la résolution de ce problème. Afin de spécifier facilement des contraintes sur le placement, Entropy

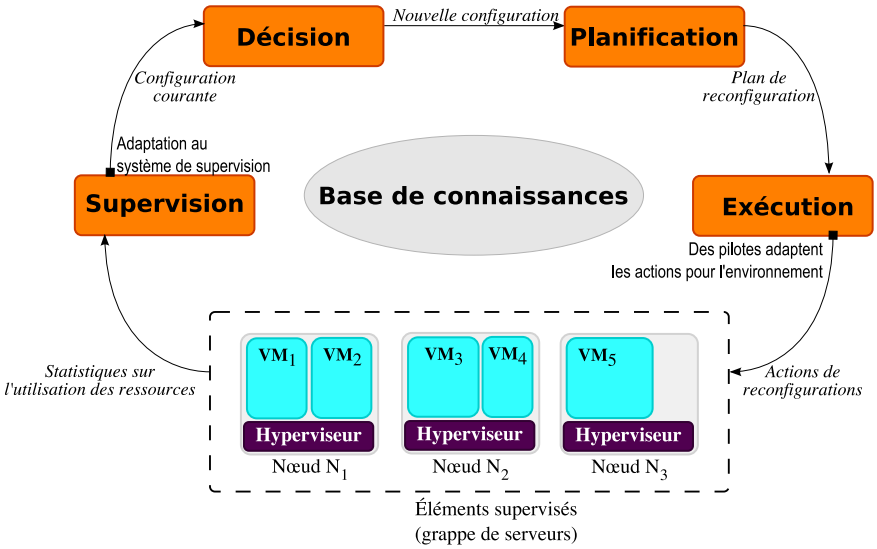


FIGURE 6.3 – Boucle de contrôle d'Entropy

met à disposition dans sa base de connaissances un ensemble de contraintes pré-implémentées et une API permettant de développer ses propres contraintes. Deux exemples d'implémentation de modules de décision seront discutés dans les chapitres 7 et 9.

Le module de planification est responsable de la détection des actions permettant l'auto-adaptation. À partir de la configuration courante et d'une configuration idéale calculée par le module de décision, ce module dresse un plan de reconfiguration décrivant un protocole d'exécution des actions assurant leur faisabilité et réduisant son temps d'application le plus possible. Ce module sera discuté en détail dans le chapitre 8.

Finalement, le module d'exécution exécute le plan de reconfiguration calculé précédemment en le décomposant en une suite d'actions qui sont envoyées aux hyperviseurs concernés. Ce module étant en contact avec l'environnement de gestion de la grappe, il doit fournir un ensemble d'adaptateurs permettant aux administrateurs de choisir une implémentation des actions compatibles avec l'environnement logiciel de la grappe.

6.2 Intégration avec l'environnement logiciel

Les concepts évoqués dans la section précédente ne sont pas spécifiques à l'environnement logiciel. Les actions manipulant les machines virtuelles durant leur cycle de vie sont disponibles avec la plupart des hyperviseurs, tandis que la notion de configuration s'abstrait de toute considération sur le système de supervision de la grappe. Il existe plusieurs systèmes de supervision, plusieurs hyperviseurs et chacun d'entre eux disposent de leur spécificités. Il est donc nécessaire de pouvoir interfacer Entropy avec des systèmes de supervision spécifiques, mais également avec des hyperviseurs spécifiques.

Nous décrivons dans cette section un exemple d'interfaçage avec le système de supervision Ganglia puis nous discutons de la méthode retenue pour interfacer Entropy avec un hyperviseur particulier.

6.2.1 Le module de supervision : exemple d'interfaçage avec Ganglia

Ganglia [MCC04] est un système de supervision de ressources distribué. Chaque nœud à superviser exécute une sonde *gmond* qui interroge le système d'exploitation afin d'extraire les différentes informations décrivant l'état des ressources de la machine, telle que la quantité de mémoire disponible et utilisée ou la distribution du temps processeur. Ces informations sont transmises à d'autres sondes appartenant à la même grappe en utilisant une adresse *multicast* ou à une sonde parente afin d'agréger les informations.

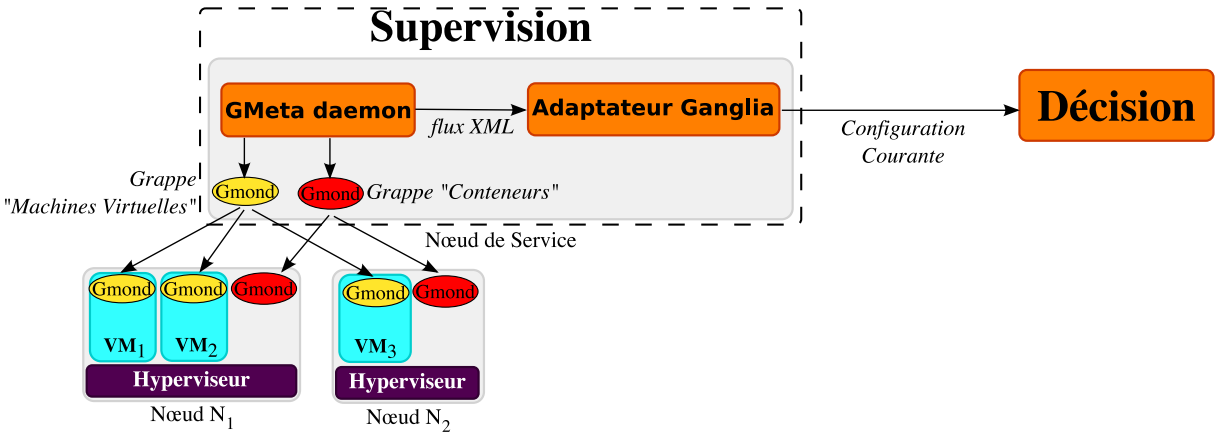


FIGURE 6.4 – Architecture du module de supervision Ganglia interfacé avec Entropy

Un deuxième composant, le *gmetad*, sert de collecteur de données. Celui-ci interroge une sonde *gmond* par grappe et stocke les informations dans une base de données au format RRD. Ce composant met à disposition une connexion TCP exportant l'état courant des ressources de toutes les grappes supervisées. Les données sont formatées au travers d'un flux XML qu'il est possible d'analyser afin d'extraire les informations nécessaires à l'établissement de la configuration courante.

La Figure 6.4 représente une architecture logicielle basée sur l'utilisation de Ganglia comme système de supervision. En plus des sondes sur les nœuds de calcul, il est nécessaire de déployer une sonde sur chaque machine virtuelle afin de remonter les informations relatives à leur consommation CPU. Nous avons choisi de considérer que les machines virtuelles et les nœuds de calcul appartenaient à deux grappes différentes. Les besoins en ressources CPU des machines virtuelles sont calculés d'après le pourcentage de temps CPU consommé par les machines virtuelles et la capacité CPU de leur nœud d'accueil.

Par défaut, les informations fournies par Ganglia ne permettent pas de déterminer la position des machines virtuelles. De plus, en fonction de l'hyperviseur, certaines informations additionnelles sont à fournir afin de combler les manques : dans le cas d'une grappe utilisant l'hyperviseur Xen [BDF+03] par exemple, la sonde s'exécute dans le « Domaine-0 » (machine virtuelle dédiée à la gestion de l'hyperviseur) et Ganglia ne peut déterminer le nombre de CPU physiques disponibles sur la machine ni la quantité de mémoire utilisable. Pour ces raisons, des métriques supplémentaires doivent être insérées dans la sonde *gmond* au travers d'un script *shell* exécuté sur chaque nœud de calcul.

6.2.2 Le module d'exécution

Le module d'exécution est chargé de la supervision et de l'exécution d'une reconfiguration par l'application d'un plan de reconfiguration fourni par le module de planification. Ce plan décrit les actions à exécuter sur les machines virtuelles. Il s'agit des transitions dans le cycle de vie des machines virtuelles. Ces actions sont considérées à ce moment comme abstraites car elles ne sont liées à aucun hyperviseur précis. Le module d'exécution étant en contact direct avec l'environnement logiciel de la grappe il doit donc transformer chaque action abstraite en une action concrète équivalente, compatible avec l'environnement.

La transformation des actions abstraites en actions concrètes est réalisée par le biais de pilotes (*drivers*). Pour chaque type d'action, l'administrateur assigne un pilote permettant de la réaliser. L'administrateur peut implémenter ses propres pilotes ou utiliser les pilotes disponibles dans la base de connaissances. Dans Entropy, des pilotes permettent d'exécuter des actions pour l'hyperviseur Xen 3.2 par le biais du serveur HTTP embarqué dans l'hyperviseur ou par l'interface XML-RPC [MSS08]. Il est également possible d'exécuter les actions par le biais de commandes *shell* exécutées dans une session SSH sur le nœud d'accueil de la machine virtuelle à manipuler.

Il est possible aussi de définir des actions concrètes plus génériques. Des efforts récents de la part des acteurs de la virtualisation tendent vers l'établissement de standards concernant la spécification des machines virtuelles et leur manipulation. Le standard OVF [OVF09] par exemple, définit un format

générique pour des machines virtuelles déployables sur tous les hyperviseurs compatibles. Libvirt [LVT] est un gestionnaire d'hyperviseur, il s'intercale au dessus de l'hyperviseur et permet de manipuler les machines virtuelles par le biais d'une interface commune quelque soit le type de l'hyperviseur sous-jacent (Xen, KVM/Qemu [Hab08], VirtualBox [VBO], OpenVZ [OVZ], LxC [LxC]). En déployant Libvirt au dessus des hyperviseurs de chaque nœud de calcul et en utilisant des pilotes compatibles, il est possible d'obtenir un environnement générique pour la manipulation des machines virtuelles dans les grappes, indépendamment de l'hyperviseur déployé sur les nœuds. Ce type d'implémentation n'a cependant pas pu être réalisé dans le cadre de cette thèse de par l'absence d'une interface générique permettant de communiquer avec Libvirt. Celui-ci est entièrement écrit dans le langage C et bien qu'un portage pour le langage JAVA existe, il repose sur l'utilisation d'un pont JNI n'assurant aucune portabilité directe de l'application. Une solution consisterait alors à développer un serveur XML-RPC pour Libvirt.

6.3 La base de connaissances

La base de connaissances d'Entropy met à disposition un ensemble d'outils pour le développement de modules de décision adaptés aux besoins des administrateurs. Dans le cadre de cette thèse, nous avons choisi d'utiliser une approche reposant sur de la programmation par contraintes. Nous décrivons dans une première partie la modélisation de l'affectation des machines virtuelles, puis nous décrivons au travers d'exemples l'API permettant d'ajouter ses propres contraintes de placement au problème de base. Les définitions des différentes contraintes dédiées aux module de décision sont disponibles en annexe de cette thèse.

6.3.1 Modélisation du problème d'affectation des machines virtuelles

Le modèle de programmation par contraintes représente une configuration impliquant n nœuds hébergeant k machines virtuelles. Le modèle définit d'abord les différents nœuds et machines virtuelles, la capacité CPU et mémoire de chaque nœud ainsi que les besoins CPU et mémoire de chaque machine virtuelle :

- Chaque nœud impliqué dans la configuration est identifié par un entier $i \in [1, n]$. Chaque machine virtuelle est identifiée par un entier $j \in [1, k]$.
- Le vecteur d'entiers $\mathcal{R}_c = \langle r_c^1, \ldots, r_c^j, \ldots, r_c^k \rangle$ désigne les besoins en ressources CPU de chaque machine virtuelle. Ainsi, r_c^j indique le besoin en ressources CPU de la machine virtuelle d'indice j. De la même manière, on définit le vecteur $\mathcal{R}_m$ pour désigner les besoins en mémoire de chaque machine virtuelle.
- Le vecteur de variables entières $\mathcal{C}_c = \langle c_c^1, \ldots, c_c^i, \ldots, c_c^n \rangle$ désigne la capacité CPU libre de chaque nœud. Ainsi, la variable entière c_c^i décrit la capacité CPU restante du nœud d'indice i. La borne supérieure de chacune de ces variables est égale à leur capacité CPU maximale et la borne inférieure est égale à 0. De la même manière, on définit le vecteur $\mathcal{C}_m$ pour désigner la capacité mémoire de chacun des nœuds.

Nous avons ensuite modélisé la relation d'affectation des machines virtuelles sur les nœuds et fonction de leur état et sa réciproque de la manière suivante :

- Le vecteur de variables entières $\mathcal{V} = \langle v_1, \ldots, v_j, \ldots, v_k \rangle$ avec $dom(v_j) = [1, n+2]$ indique l'indice du nœud hébergeant chaque machine virtuelle. Ainsi $v_j = i$ indique que la machine virtuelle d'indice j est hébergée sur le nœud d'indice i. Les nœuds d'indice $n+1$ et $n+2$ sont des nœuds virtuels permettant de spécifier l'état de la machine virtuelle si nécessaire. Ainsi, si $v_j \in [1, n]$ alors la machine virtuelle d'indice j sera nécessairement dans l'état `Exécution`, si $v_j = n+1$ alors la machine virtuelle sera nécessairement dans le pseudo état `Prêt`, si $v_j = n+2$ alors la machine virtuelle sera nécessairement dans l'état `Terminé`.
- Inversement, pour chaque nœud d'indice i (y compris les 2 nœuds virtuels), le vecteur de variables booléennes $H_i = \left\langle h_i^1, \ldots, h_i^j, \ldots, h_i^k \right\rangle$ précise les machines virtuelles que celui-ci héberge. Ainsi, si la machine virtuelle d'indice j est hébergée sur le nœud d'indice i, alors $h_i^j = 1$ sinon $h_i^j = 0$.

On observe que les variables v_j et h_i^j sont sémantiquement équivalentes. Afin d'assurer la consistance entre la représentation de la relation d'affectation et sa réciproque, nous avons défini la contrainte d'intégrité décrite par l' équation (6.1).

$$\forall i \in [1, n+2], \forall j \in [1, k], v_j = i \Leftrightarrow h_i^j = 1 \tag{6.1}$$

Nous appelons ce problème de satisfaction de contraintes (CSP) basique VMAP (*Virtual Machine Assignement Problem*). Résoudre ce problème consiste à affecter une valeur à chaque variable de l'ensemble $\mathcal{V}$. Nous pouvons alors déduire de cette affectation une configuration. Si la valeur d'une variable correspond à l'indice d'un nœud alors la machine virtuelle sera hébergée dans l'état `Exécution` sur ce nœud. Si la valeur de la variable correspond à la valeur $n+2$ alors la machine virtuelle sera dans l'état `Terminé` et celle-ci n'apparaîtra pas dans la configuration. Finalement, si la valeur de la variable est égale à $n+1$ alors son état courant est considéré pour décider de son état final. Si la machine virtuelle était dans l'état `Exécution` alors la configuration la déclarera dans l'état `Endormi`. Sinon, elle sera dans l'état `Attente` et restera hébergée sur son nœud courant.

6.3.2 Une API pour contraindre le placement des machines virtuelles

Lorsqu'un utilisateur veut spécialiser le placement des machines virtuelles ou l'ordonnancement des tâches, celui-ci développe son propre module de décision en spécialisant le VMAP afin de calculer une configuration répondant à ses besoins. Dans ce contexte, il convient donc de traduire les besoins sous la forme de contraintes.

Nous avons ainsi développé une API permettant d'ajouter des contraintes de placement et d'ordonnancement au dessus du solveur de contraintes Choco [CJLR06]. Ce solveur met à disposition une bibliothèque JAVA permettant la résolution de problèmes de satisfactions de contraintes de d'optimisation. La liste des contraintes développées dans le cadre de cette thèse est disponible en annexe mais l'ensemble peut cependant être enrichi au besoin. Ces contraintes permettent de satisfaire les utilisateurs des machines virtuelles qui souhaitent que le placement de leurs machines virtuelles respecte une certaine topologie ou des demandes d'administrateurs qui souhaitent ajouter des règles de sécurités, de fragmentation des nœuds ou d'ordonnancement.

Le Listing 6.1 présente un exemple d'utilisation de l'API d'Entropy et de la base de connaissances en posant des contraintes de placement sur différents ensembles de machines virtuelles et de nœuds. On considère dans cet exemple l'hébergement de deux tâches (`job1` et `job2`) sur quatre nœuds (`n1` à `n4`). La tâche `job1` héberge une application scientifique distribuée critique : la machine virtuelle `master` envoie en plusieurs exemplaires des calculs aux applications des machines virtuelles `slave1` et `slave2`. L'administrateur souhaite d'abord exécuter cette tâche sur la grappe, il contraint donc la configuration résultat en précisant que les machines virtuelles devront être dans l'état `Exécution` (ligne 6 à ligne 8). Le propriétaire de cette tâche souhaite une certaine tolérance aux pannes sur les machines virtuelles esclaves regroupées dans l'ensemble `slaves_job1`. Il est donc nécessaire d'héberger ces machines virtuelles sur des nœuds différents. Cette contrainte est spécifiée à la ligne 9 du Listing. À la ligne 10, nous demandons à ce que la machine virtuelle `master` soit nécessairement hébergée sur le nœud `n1`.

La tâche `job2` exécute une application distribuée sur deux machines virtuelles. Les deux composants de l'application étant fortement inter-communicant et pour des raisons de performances, si ces machines virtuelles ne sont pas gourmandes en ressources, il est possible de les regrouper explicitement sur un même nœud afin de ne pas limiter les performances de l'application par la bande passante ou la latence du réseau. Cette contrainte est exprimée à la ligne 12.

Le Listing présente également des contraintes qui concernent les administrateurs de la grappe. Ainsi, les lignes 14 à 16 spécifient que chaque nœud hébergera au plus deux machines virtuelles. Les lignes 17 à 19 déclarent que les machines virtuelles de la tâche `job2` doivent être en état `Exécution` et qu'elles ne devront pas être hébergées sur le nœud `n4`.

```
ManagedElementSet <VirtualMachine>job1; // = {master, slave1, slave2}
ManagedElementSet <VirtualMachine>slaves_job1; // = {slave1, slave2}
ManagedElementSet <VirtualMachine>job2; // = {c1, c2}
```

```
ManagedElementSet <Node>nodes; // = {n1, n2, n3, n4}

for (VirtualMachine vm : job1) {
    model.mustBeRunning(vm);
}
model.assignOnDifferentNodes(slaves_job1);
model.restrictOnNode(job1.get("master"),nodes.get("n1"));

model.assignOnSameNode(job2);

for (Node n : nodes) {
    model.setMaximumVirtualMachinesOnNode(n, 2);
}
for (VirtualMachine vm : job2) {
    model.mustBeRunning(vm);
    model.denyVirtualMachineOnNode(vm, nodes.get("n4"));
}
model.solve(10);
Configuration result = model.getResultingConfiguration();
```

Listing 6.1 – Spécifications de contraintes de placement dans Entropy

Finalement, à la ligne 21, nous demandons au solveur de contraintes de calculer une configuration satisfaisant toute les contraintes dans un temps inférieur à 10 secondes puis nous récupérons la configuration résultat à la ligne 22. La Figure 6.5 décrit une configuration résultat possible.

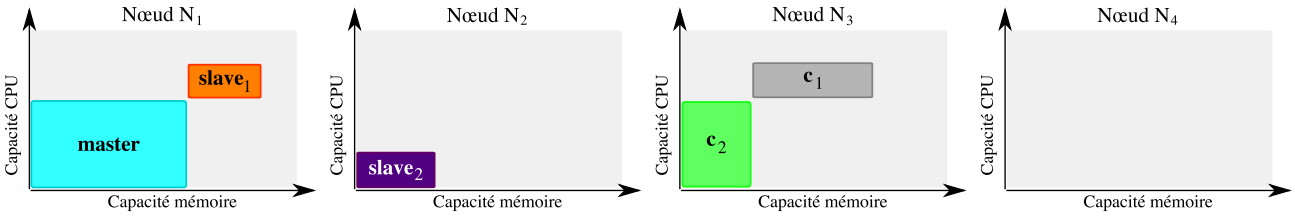


FIGURE 6.5 – Une configuration résultat satisfaisant les contraintes du Listing 6.1

6.4 Travaux apparentés

En 2003, Chase *et al.* [CIG+03] proposent la notion de « grappe à la demande » , un ensemble de mécanismes pour l'instantiation à la volée de machines virtuelles ou de nœuds de calcul afin de fournir aux utilisateurs un ensemble de machines physiques ou virtuelles interconnectées. Ces travaux ont démontré l'intérêt d'obtenir à la demande des environnements personnalisés et des ressources. L'environnement Violin [RRX+06, RJXG05] reprend ces différents travaux et propose une approche reposant sur l'informatique autonome afin d'adapter le placement de machines virtuelles dans les grappes en fonction de leurs besoins. Cette auto-optimisation, reposant sur une heuristique de placement ad-hoc, évite de surcharger les nœuds de calculs afin d'obtenir en continu de bonnes performances.

La mise en avant récente des architectures de type « nuages d'ordinateurs » (*cloud computing*) a permis le développement rapide de plusieurs environnements de gestion de machines virtuelles dans des infrastructures distribuées. Usher [MGVV07] propose un environnement manipulant des machines virtuelles dans des grappes utilisant l'hyperviseur Xen. Une invite de commande assure la gestion des machines virtuelles et des nœuds de manière interactive ou par le biais de scripts, tandis qu'un ensemble de greffons permet à Usher de s'interfacer avec différents modules dédiés à l'administration de la grappe.

Les environnements logiciels Eucalyptus [NWG+08], OpenNebula [Mon08], Nimbus [nim] ou Shirako [GIYC06] proposent une approche orientée services web pour la gestion de parc de machines virtuelles durant tout leur cycle de vie. Ces différents environnements embarquent une quantité de services

standards variables comme des services d'authentification, des services DHCP ou des serveurs de fichiers afin d'installer l'application sans pré-requis logiciels. On note la possibilité d'utiliser plusieurs types d'hyperviseurs, soit par le biais d'API, soit en utilisant la couche d'abstraction fournie par Libvirt.

Ces solutions proposent une API et une architecture modulaire dédiées au développement de modules de décision spécifiques. Grit *et al.* proposent similairement à Entropy, avec une extension de Shirako une séparation entre la partie décisionnelle et les mécanismes permettant de réaliser l'adaptation. Les différentes API misent à disposition se limitent cependant à l'écriture d'heuristiques de placement ou d'ordonnancement standard. OpenNebula propose en sus un moteur de règles permettant aux utilisateurs et aux administrateurs de spécifier des contraintes sur le placement des machines virtuelles. Cette approche est cependant limitée par les soucis de composition de règles et de conflits que nous avons évoqués dans le chapitre 5.

Cette thèse se focalise sur la partie décisionnelle et la planification de l'adaptation du placement des machines virtuelles. À l'opposé, les différents environnements exposés ci-dessus se focalisent sur la définition d'une architecture permettant la manipulation des machines virtuelles. En ce sens, ils sont complémentaires et il semble possible d'interfacer Entropy avec ces différents environnements en développant des adaptateurs pour les systèmes de supervision de ces logiciels et des pilotes pour l'exécution des actions. Cette intégration faciliterait le déploiement du module de décision, de planification et de la base de connaissances d'Entropy ainsi que sa maintenance.

6.5 Conclusion

Bilan

Nous avons présenté dans ce chapitre l'architecture générale du logiciel Entropy qui a été développé au cour de cette thèse. Il permet une auto-optimisation en continu du placement de machines virtuelles et de l'ordonnancement des tâches en agissant sur leur cycle de vie. Les interfaces mises à disposition assurent de pouvoir utiliser Entropy avec des environnements logiciels spécifiques : le développement d'adaptateur pour les systèmes de supervision usuels tel que Ganglia permet d'extraire la configuration courante tandis que l'utilisation de pilotes permet d'exécuter les actions manipulant les machines virtuelles sur un hyperviseur spécifique tel que Xen. La séparation de la partie décisionnelle calculant un nouveau placement idéal et des mécanismes de mise en place de cet agencement permet aux administrateurs de développer leur propre module de décision en utilisant l'API bâtie autour du solveur de contraintes Choco.

Perspectives

Notre approche propose un environnement complet permettant de contrôler les machines virtuelles durant la totalité de leur cycle de vie. Cependant la contribution de cette thèse se focalise sur l'aspect décisionnel et la planification de l'auto-adaptation. La nécessité de maintenir la totalité d'un environnement de gestion des machines virtuelles est potentiellement un frein au déploiement d'Entropy. Utiliser des machines virtuelles dans les grappes nécessite en effet de revoir l'architecture de celles-ci, notamment au niveau du stockage ou des plans d'adressage du réseaux. Simultanément, des suites logicielles comme OpenNebula proposent de faciliter la mise en place d'environnements logiciels pour grappe à base de machines virtuelles. Afin de favoriser l'utilisabilité d'Entropy, il serait important de choisir après une étude des différents environnements disponibles, un environnement de référence servant de support à notre contribution.

Chapitre 7

Consolidation dynamique de machines virtuelles

Où nous discutons de la modélisation et de l'optimisation d'un module de décision réalisant de la consolidation dynamique de machines virtuelles. Cette approche permet de concentrer les machines virtuelles en cours d'exécution sur un nombre minimum de nœuds tout en satisfaisant des besoins en ressources CPU et mémoire variables. Cette stratégie de placement dynamique permet d'augmenter la capacité d'accueil de la grappe en hébergeant simultanément plus de machines virtuelles ou de réduire la consommation énergétique de la grappe en éteignant les nœuds inutilisés.

Sommaire

L'observation du taux moyen d'utilisation des grappes de serveurs met en évidence une sous-utilisation de ces infrastructures. En effet, le site *Parallel Workload Archive* [PWA] met en évidence un taux d'utilisation moyen de $53,44\%$ pour les 23 traces d'activité renseignant cette information. L'analyse de ces traces montre cependant que les grappes ne sont pas soumises en permanence à un niveau d'utilisation constant. Elles alternent le plus souvent des pics d'activités occupant la quasi-totalité des nœuds de calcul avec des périodes calmes où la grappe est grandement sous-utilisée. Ainsi, malgré un taux d'utilisation très variable des grappes, l'architecture reste disponible et opérationnelle intégralement. Les coûts de fonctionnement de la grappe tels que la consommation électrique ou les frais de climatisation ne sont pas liés à son utilisation réelle. Une partie de ces coûts n'est alors pas rentabilisée.

La consolidation consiste à héberger plusieurs machines virtuelles sur un même nœud. En éteignant les nœuds inutilisés, cette approche réduit la consommation énergétique des grappes. Elle augmente également la capacité d'hébergement d'une grappe en permettant d'héberger simultanément plus de machines virtuelles. Cependant, si l'allocation des ressources pour les machines virtuelles est dynamique, il peut être nécessaire de revoir cet agencement en fonction des besoins courants afin de maintenir un niveau de performance optimale.

La consolidation pour dans le but d'éteindre les nœuds inutilisés est un dérivé de la stratégie VOVO (*Vary On Vary Off*) [CIG+03, PBCH02] où un répartiteur de charges redirige des requêtes HTTP sur un nombre minimum de serveurs pour éteindre les nœuds inutilisés. En fonction de la charge, des nœuds supplémentaires peuvent être allumés pour maintenir un niveau de performances correcte. Dans le cadre

de la consolidation dynamique, les machines virtuelles peuvent exécuter des applications nécessitant un long temps de calcul ou des applications à haute disponibilité. Il n'est donc pas envisageable d'attendre la fin de l'exécution d'une application pour rééquilibrer la charge. Dans cette situation, la solution consiste à migrer à chaud les machines virtuelles en cours d'exécution pour concentrer la charge sur un nombre réduit de nœuds. Cette approche s'apparente à une gestion dynamique de tâches où les machines virtuelles appartenant aux tâches en cours d'exécution sont assimilées à des tâches évolutives (les besoins en ressources varient naturellement durant l'exécution) et la migration à chaud des machines virtuelles dans l'état `Exécution` permet de réaliser un partitionnement spatial dynamique des ressources en fonction des besoins courants en mémoire et en CPU.

Dans ce chapitre, nous discutons de la modélisation d'un problème d'optimisation à base de contraintes calculant une configuration utilisant un minimum de nœuds tout en satisfaisant les besoins en ressources des machines virtuelles en cours d'exécution. Nous discutons ensuite des différentes stratégies que nous avons considérées pour réduire le temps de résolution du problème. Dans une troisième section, nous présentons l'implémentation de ce problème dans Entropy puis nous positionnons notre approche par rapport à différents travaux apparentés. Finalement, nous concluons ce chapitre en discutant des limites de notre approche et de ses extensions possibles.

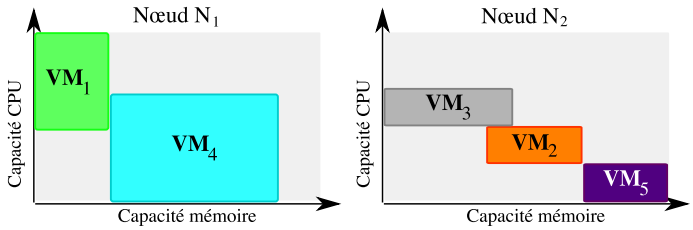
7.1 Modélisation

Le problème basique VMAP définit dans le Chapitre 6 calcule des configurations basiques sans considération des besoins en ressources CPU et mémoire des machines virtuelles ni du maintien de l'état des machines virtuelles. Dans le cas de la consolidation dynamique, il importe d'abord de s'assurer que les différentes machines virtuelles ne changeront pas d'état durant l'auto-optimisation et que les machines virtuelles dans l'état `Exécution` auront accès à suffisamment de ressources. Cette donnée est primordiale si l'on souhaite obtenir une configuration où le niveau de performance des machines virtuelles est optimal. Résoudre un tel problème s'apparente alors à une stratégie de placement dynamique des machines virtuelles.

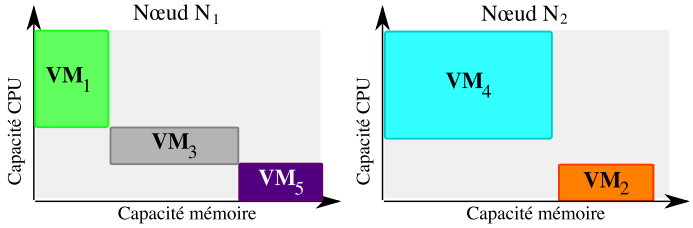
Une configuration est définie comme viable si la somme des besoins en ressources CPU et mémoire de toutes les machines virtuelles hébergées sur chaque nœud est inférieure ou égale à la capacité du nœud. La Figure 7.1(a) représente une configuration non viable où le nœud N_1 ne peut satisfaire simultanément les besoins CPU des machines virtuelles VM_1 et VM_4. On observe sur le nœud N_2 une situation similaire mais concernant la ressource mémoire. Dans ces situations, les ressources sont partagées entre les différentes machines virtuelles des nœuds et celles-ci peuvent ne pas avoir accès à suffisamment de ressources pour fonctionner à un niveau de performance optimal. Une configuration viable est décrite sur la Figure 7.1(b).

Si la surcharge CPU est une situation commune, les possibilités de surcharger la mémoire d'un nœud physique sont soumises au mode de fonctionnement de l'hyperviseur. Sur les hyperviseurs ne permettant pas de partager de pages mémoires entre différentes machines virtuelles, l'utilisateur spécifie une quantité de mémoire à allouer à sa machine virtuelle. Si cette quantité est disponible alors l'hyperviseur alloue la mémoire et lance la machine virtuelle. Cette quantité peut être réduite mais ne peut en aucun cas être supérieure à sa valeur initiale. La surcharge mémoire n'est alors pas possible. Cette situation est par contre envisageable lors de l'utilisation d'hyperviseurs partageant la mémoire tel que VMWare ESX Server [Wal02] ou Xen 3.3 [GLV+08]. En partageant des pages mémoires communes entre plusieurs machines virtuelles, il est possible de supporter un ensemble de machines virtuelles ayant des besoins en mémoire supérieurs à la capacité du nœud. Cependant, si le partage n'est plus suffisant, alors l'hyperviseur doit utiliser de la mémoire dans la zone d'échange ou refuser toute demande d'allocation supplémentaire de mémoire pour les machines virtuelles.

Ces besoins d'accès en ressources, caractérisant une configuration viable, s'expriment facilement par des contraintes. L'équation (7.1) définit des contraintes assurant l'accessibilité des besoins CPU et mémoire sur chaque nœud d'indice i en se basant sur le modèle défini par le VMAP. Chacune de ces contraintes est une contrainte globale appelée « sac-à-dos » [MT90] (*knapsack*). Cette contrainte est une contrainte générique dans le solveur Choco. Pour résoudre ce problème NP-Complet, le solveur évalue dynamiquement la quantité de ressources restante sur chaque nœud en utilisant de la programmation dynamique [Tri01], c'est à dire en calculant la solution d'un sous-problème puis d'un problème plus grand, jusqu'au problème complet.



(a) Une configuration non viable



(b) Une configuration viable

FIGURE 7.1 – Exemple de deux configurations

$$\begin{aligned} \mathcal{R}_c \cdot H_i &\leq c_c^i \quad \forall i \in [1,n] \\ \mathcal{R}_m \cdot H_i &\leq c_m^i \quad \forall i \in [1,n] \end{aligned} \tag{7.1}$$

En ajoutant les contraintes de sac-à-dos définies, nous assurons que toute les solutions calculées par le solveur Choco seront nécessairement des configurations viables.

Nous définissons ensuite la variable X comptabilisant le nombre de nœuds hébergeant au moins une machine virtuelle en état `Exécution`. Dans cette situation, la variable X est une variable objectif dont on souhaite minimiser la valeur jusqu'à sa valeur minimale x_{vmpp}. Sa valeur est définie dans l'Équation (7.2).

$$X = \sum_{i \in [1,n]} u_i, \text{ où } u_i = \begin{cases} 1, \text{si } \exists j \in [1,k] \mid h_i^j = 1 \\ 0, \text{sinon} \end{cases}$$

$$x_{vmpp} = min\,(X) \tag{7.2}$$

Nous désignons par VMPP (*Virtual Machines Packing Problem*) le problème du calcul de la valeur minimale de X. Il s'agit en fait du problème d'optimisation NP-Difficile *2-D Bin Packing* [Sha04] où les deux dimensions correspondent aux ressources CPU et mémoire des nœuds qui sont partagées entre les machines virtuelles en cours d'exécution.

7.2 Optimisation de la résolution

Le VMPP définie dans la section précédente est solvable tel quel par un solveur de contraintes. Cependant, la complexité de ce problème peut rendre le processus de résolution lent sur des problèmes de grandes tailles. Il importe alors d'améliorer le processus de résolution en utilisant les principes de filtrage et d'heuristiques de branchement décris dans le chapitre 4.

7.2.1 Estimation du nombre de nœud minimum

Afin de limiter le temps de calcul de la solution optimale x_{vmpp}, nous avons réduit le domaine de la variable X en définissant des bornes les plus proches possibles de x_{vmpp} par le biais d'heuristiques ou d'algorithmes calculant une valeur approchée rapidement. L'équation (7.3) définit la borne supérieure de

la variable X. L'heuristique *First Fit Decrease* (FFD) [CGJ96] calcule une valeur approchée par le dessus de x_{vmpp}. Cette valeur x_{ffd} est calculée en affectant chaque machine virtuelle sur le premier nœud disposant de suffisamment de ressources, en commençant par les machines virtuelles les plus consommatrices. L'heuristique FFD ne permet pas nécessairement de calculer une solution à un problème mais si elle existe. Pour palier à cette éventualité, en cas d'échec de l'heuristique nous définissons la borne supérieure de X en considérant le nombre de nœuds n et de machines virtuelles k.

$$X \leq \begin{cases} x_{ffd} \\ \min(n,k), \text{sinon} \end{cases} \tag{7.3}$$

La borne inférieure de la variable X est définie en fonction des nœuds de la grappe. L'équation (7.4) définit cette borne si l'ensemble des nœuds de calcul n'est pas homogène, c'est à dire s'ils ne disposent pas tous de capacités mémoire et CPU identiques. Dans ce cas, nous sommons les besoins CPU des machines virtuelles et nous définissons la valeur x_{cpu} comme le nombre de nœuds nécessaires pour répartir ces besoins. La valeur x_{mem} considérant les ressources mémoire est calculée de la même manière. Si l'ensemble des nœuds de calculs est homogène, alors la borne inférieure est définie par l'équation (7.5).

$$X \geq \max(x_{cpu}, x_{mem}) \tag{7.4}$$

$$X \geq \max\left(\left\lceil \frac{\sum_{j\in[0,k]} r_c^j}{c_c^i} \right\rceil, \left\lceil \frac{\sum_{j\in[0,k]} r_m^j}{c_m^i} \right\rceil\right), i \in [0,n] \tag{7.5}$$

7.2.2 Utilisation des symétries

Si l'ensemble des nœuds de calcul n'est pas nécessairement homogène, certains nœud peuvent tout de même être identiques. Il en va de même pour les machines virtuelles qui a un moment donné peuvent avoir les mêmes besoins mémoire et CPU. Les équations (7.6) et (7.7) définissent une relation d'équivalence respectivement entre les nœuds et les machines virtuelles. Cette considération permet d'améliorer le processus de résolution du VMPP en rajoutant simplement une contrainte au modèle qui procède à un filtrage supplémentaire de l'arbre de recherche en supprimant les affectations symétriques : si un nœud d'indice a ne peut héberger la machine virtuelle d'indice x (la valeur a n'est pas dans le domaine de la variable v_x), alors il ne peut héberger aucune machine virtuelle équivalente. De la même manière, la machine virtuelle d'indice x ne peut pas être hébergée sur n'importe quelle nœud équivalent au nœud d'indice a. L'équation (7.8) formalise cette contrainte en reprenant le modèle du VMAP.

$$\forall a,b \in [0,n], \mathrm{N}_a \equiv \mathrm{N}_b \Leftrightarrow c_c^a = c_c^b \wedge c_m^a = c_m^b \tag{7.6}$$

$$\forall x,y \in [0,k], \mathrm{VM}_x \equiv \mathrm{VM}_y \Leftrightarrow r_c^x = r_c^y \wedge r_m^x = r_m^y \tag{7.7}$$

$$\begin{aligned} &\forall a,b \in [0,n], \forall x,y \in [0,k], \mathrm{VM}_x \equiv \mathrm{VM}_y, \mathrm{N}_a \equiv \mathrm{N}_b, \\ &a \notin \mathcal{D}(v_x) \Rightarrow a \notin \mathcal{D}(v_y) \wedge b \notin \mathcal{D}(v_x) \wedge b \notin \mathcal{D}(v_y) \end{aligned} \tag{7.8}$$

7.2.3 Heuristiques de branchement

Nous avons expliqué dans le chapitre 4 que l'utilisation d'heuristiques de branchement adaptées permet d'accélérer la résolution d'un problèmes d'optimisation en guidant la recherche vers des solutions critiques ou prometteuses.

Pour ce problème, nous avons définit une heuristique de choix de variable sélectionnant en priorité les machines virtuelles ayant les besoins en ressources mémoire les plus importants. Cette heuristique

permet d'affecter les machines virtuelles critiques au plus tôt. L'heuristique de choix de valeur quant à elle affecte en premier la machine virtuelle au nœud qui l'héberge afin de limiter le nombre de migrations. Si cette affectation n'est pas possible alors le solveur teste l'affectation de la machine virtuelle sur le nœud disposant de la plus petite quantité de mémoire disponible. On retrouve ici une approche *first fail* permettant de détecter au plus tôt les affectations inconsistantes afin d'élaguer l'arbre de recherche le plus tôt possible.

7.3 Implémentation dans Entropy

L'ensemble des contraintes définissant le VMPP ainsi que différentes optimisations accélérant le temps de calcul de la solution ont été implémentées dans Entropy en utilisant le solveur de contraintes Choco. Le Listing 7.1 décrit le code Java de l'implémentation de VMPP dans Entropy en omettant le calcul des bornes de X et les informations de deboggages. Nous instantions un nouveau solveur à la Ligne 7 en lui indiquant la configuration courante, nécessaire à la création du problème de base VMAP puis nous définissons la variable d'optimisation X. Ligne 10 à 18 nous ajoutons des contraintes indiquant que l'état de chaque machine virtuelle doit être identique à l'état courant. Des lignes 21 à 29, nous définissons les contraintes assurant la minimisation de la variable X. La contrainte $AtMostNValue$ décrit que les variables d'affectations de l'ensemble $\mathcal{V}$ doivent avoir un nombre de valeurs différentes minimum. Chaque valeur dénotant l'indice du nœud hébergeant une machine virtuelle, nous demandons alors d'héberger les machines virtuelles sur le nombre minimum de nœuds. Les contraintes $LBPDyn$ calculent dynamiquement une borne inférieure de X, non nécessairement réalisable. La ligne 30 déclare l'utilisation de la contrainte $SymetryBreakingVMPP$ qui réalise un filtrage supplémentaire en considérant les machines virtuelles et les nœuds équivalents. Nous déclarons ensuite les contraintes de sac à dos pour tout les nœuds puis nous indiquons les heuristiques de branchement pour la résolution de ce problème. Finalement, nous demandons à la ligne 39 de minimiser la valeur de la variable X en précisant un délai maximal.

```
public class VirtualMachinesPackingProblem extends DecisionModule {

3       ...
        public Configuration compute() throws AssignmentException, Exception {
        Configuration curConf = this.getCurrentObservations().getConfiguration();

        ChocoSolver model = new ChocoSolver(curConf);
8       IntDomainVar X = ... ;

        for (VirtualMachine vm : curConf.getRunnings()) {
         model.mustBeRunning(vm);
        }
13      for (VirtualMachine vm : curConf.getSleepings()) {
         model.mustBeReady(vm);
        }
        for (VirtualMachine vm : curConf.getWaitings()) {
         model.mustBeReady(vm);
18      }


        model.addConstraint(new AtMostNValue(model.getAssignmentVars(), X));
        model.addConstraint(new LBPdyn(model.getAssignmentVars(),
23              model.getNodeResources(Node.CPU_CAPACITY),
                model.getVirtualMachineResourcesValue(VirtualMachine.CPU_CONSUMPTION),
                X));
        model.addConstraint(new LBPdyn(model.getAssignmentVars(),
                model.getNodeResources(Node.MEMORY_TOTAL),
28              model.getVirtualMachineResourcesValue(VirtualMachine.MEMORY),
                X));
        model.addConstraint(new SymetryBreakingVMPP(model));
```

```
        for (Node node : curConf.getOnlines()) {
33       model.setKnapsackOn(node, VirtualMachine.CPU_CONSUMPTION, Node.CPU_CAPACITY);
         model.setKnapsackOn(node, VirtualMachine.MEMORY, Node.MEMORY_TOTAL);
        }

        model.setVirtualMachineSelector(new BiggestUnassignedVirtualMachine(model, X,
            VirtualMachine.MEMORY));
38      model.setNodeSelector(new BusiestNode(model, X, Node.MEMORY_TOTAL, true));
        if (!model.minimize(X, this.getTimeout())) {
         logger.debug("No solution available");
         throw new AssignmentException(curConf);
        }
43      return model.getResultingConfiguration();
        }
    }
```

Listing 7.1 – Extrait de l'implémentation du problème VMPP dans Entropy

7.4 Travaux apparentés

Différents travaux ont abordé la problématique de la consolidation des machines virtuelles. Nous décrivons dans cette section ces travaux apparentés en considérant les différents critères pris en compte pour la consolidation et les différents algorithmes calculant une nouvelle configuration.

Khanna *et al.* [KBKK06] proposent une approche minimisant les portions de ressources inutilisées. Cette approche est basée sur une heuristique *Best Fit Decrease*, une extension de l'heuristique FFD qui affecte chaque machine virtuelle sur le nœud dont la capacité libre en ressources est la plus proche possible des besoins de la machine virtuelle. Cette approche améliore sensiblement la qualité de certaines solutions mais tend à laisser de petits espaces libres sur chaque nœuds, cette fragmentation pouvant dégrader la qualité de la solution dans certaines situations. Bobroff *et al.* [BKB07] calculent la nouvelle configuration viable avec l'heuristique FFD. Un système de prédiction place les machines virtuelles en fonction de leurs besoins durant le prochain intervalle de temps. Celui-ci représente le temps de migration de la machine virtuelle d'après estimation. Cela permet d'envisager un système prévenant les configurations non viables plutôt qu'un système réalisant uniquement la correction de la configuration courante. En intercalant un tel module entre le module de supervision et le module de décision d'Entropy, il est possible de reproduire un tel système. Ces deux approches permettent d'obtenir des configurations viables utilisant un nombre de nœuds proche de la solution optimale. Cependant, en ne considérant pas la configuration courante, le nombre de migrations à réaliser est proche du nombre de machines virtuelles impliquées dans la configuration et entraîne un temps de reconfiguration dégradant fortement l'intérêt de l'approche.

Verma *et al.* [VAN08b], Wood *et al.* [WSVY07] et le prototype pMapper [VAN08a] considèrent la configuration courante afin de réduire le nombre de migrations à réaliser. Leurs solutions utilisent une variation de l'algorithme FFD et distinguent les nœuds surchargés des nœuds sous-utilisés. L'algorithme sélectionne alors des machines virtuelles hébergées sur des nœuds surchargés pour les migrer sur des nœuds sous-chargés. Cette solution réduit le nombre de migrations à réaliser mais dégrade la qualité de la solution en ne considérant pas tous les déplacements de machines virtuelles possibles. En effet, en migrant également certaines machines virtuelles entre des nœuds surchargés ou sous-chargés, il peut être possible de calculer de meilleures configurations, utilisant un nombre de nœuds mais également un nombre de migrations inférieur.

Globalement, les différents travaux présentés ci-dessus proposent des solutions au problème de *Bin Packing* à 2 dimensions par le biais d'heuristiques. Les approches reposant sur les heuristiques FFD ou BFD produisent des solutions correctes, mais non nécessairement optimales, et entraînant un nombre de migration parfois très important. Exécuter ces différentes migrations peut prendre un temps significatif durant lequel les besoins des machines virtuelles pourront avoir changé, rendant alors l'auto-optimisation obsolète. Simultanément, les approches considérant la configuration courante permettent de réduire le nombre

de migrations mais dégradent la qualité de la solution (le nombre de nœuds utilisés). Ces approches ne disposent d'aucune flexibilité et la considération de nouvelles contraintes spécifiques à la grappe ou même à des machines virtuelles nécessite de revoir entièrement les heuristiques. L'approche exacte décrite dans ce chapitre calcule des configurations optimales nécessitant un nombre de migrations réduit. Il est possible de spécialiser le problème au besoin par l'ajout de nouvelles contraintes de placement, sans remettre en cause les précédentes contraintes. En contrepartie, le calcul d'une configuration est généralement plus long ; nous détaillerons cette comparaison durant l'évaluation de notre prototype dans le chapitre 10.

Nathuji *et al.* [NS07, NS08] proposent une infrastructure permettant une gestion coordonnée de stratégies d'économie d'énergie globale, par de la consolidation dynamique et locale, en adaptant la fréquence de fonctionnement des processeurs aux besoins [EKR02]. Un moniteur observe les requêtes ACPI des différentes machines virtuelles et propose un placement réduisant le nombre de nœuds allumés et si possible la fréquence de fonctionnement des processeurs. On retrouve également cette stratégie dans les travaux de Verma *et al.* [VAN08b] qui se focalisent sur des applications de type calculs haute performance. Ces deux approches diffèrent légèrement de notre approche en déclarant un objectif qui se focalise ouvertement sur la notion d'énergie alors que notre cas d'étude se focalise sur la réduction du nombre de nœuds utiles.

Wood *et al.* [WTLS+09] utilisent des hyperviseurs partageant des pages mémoires entre machines virtuelles afin d'améliorer le niveau de consolidation. En plus de calculer une configuration viable, ils considèrent le « taux d'affinité mémoire » des machines virtuelles en regroupant si possible sur un même nœud les machines virtuelles partageant le plus de pages mémoires. Chaque nœud peut alors héberger plus de machines virtuelles.

7.5 Conclusion

Bilan

Nous avons présenté dans ce chapitre un module de décision pour Entropy réalisant de la consolidation dynamique de machines virtuelles. Cette approche permet de réduire les coûts de fonctionnement des grappes en hébergeant les machines virtuelles en cours d'exécution sur un nombre minimum de nœud. En fonction des besoins courant en ressources CPU et mémoires des machines virtuelles, la configuration utilise le nombre de nœud minimum permettant de satisfaire les besoins en ressources de chacune. Notre approche reposant sur la programmation par contraintes permet de calculer une solution optimale pour ce problème en réduisant le nombre de migrations à réaliser. Contrairement aux approches heuristiques, il est possible d'enrichir VMPP en spécifiant de nouvelles contraintes ou de prendre en compte de nouveaux objectifs. En reprenant la fonction définissant l'affinité des machines virtuelles définie dans [WTLS+09], nous pouvons rapprocher les machines virtuelles de façon à maximiser le partage de la mémoire afin d'améliorer la consolidation.

Perspectives

Nous pensons que différentes optimisations permettrait d'améliorer la résolution de ce problème. Premièrement, l'estimation des bornes de la variable décrivant le nombre de nœuds est basée sur des heuristiques. Si le calcul d'une bonne borne supérieure permet d'assurer de trouver une solution rapidement, il est impératif de définir une borne inférieure la plus proche possible de la solution optimale. Une recherche plus poussée d'algorithmes calculant une borne inférieure permettrait ainsi de réduire l'espace de recherche.

L'objectif du problème VMPP est de réduire le nombre de nœuds utiles. En considérant également la réduction du nombre de migrations, nous cherchons à résoudre un problème selon deux objectifs. La résolution d'un tel problème consiste à calculer un compromis entre plusieurs valeurs objectif. Il importe alors de définir une fonction pondérant chaque objectif. Des évaluations de précédentes implémentations de ce module ont cependant montrées que le temps de résolution d'un tel problème ainsi que la qualité du résultat étaient moins satisfaisants que la résolution d'un problème ayant comme objectif le calcul du nombre minimum de nœuds et réduisant le nombre de migrations par une simple heuristique de branchement.

L'architecture actuelle du module de décision maintien l'état de toute les machines virtuelles lors du calcul de la nouvelle configuration viable, aucune tâche en attente ne pourra alors être exécutée. De

plus, notre approche considère des tâches évolutives ainsi que des partitions dynamiques dont la taille est adaptée aux besoins CPU des machines virtuelles. Nous ne pouvons donc prédire avec exactitudes leurs besoins et assurer ainsi que le VMPP aura une solution : si les besoins CPU des machines virtuelles sont trop important, alors il n'existera pas de configuration viable et les performances des applications seront dégradées jusqu'à ce que les besoins CPU baissent et qu'une nouvelle solution existe. Une solution consiste à suspendre des tâches en cours d'exécution temporairement pour maintenir une configuration viable. Ce point sera traité dans le chapitre 9.

Le module de décision est chargé du calcul d'une configuration viable décrivant la solution que l'on souhaite obtenir. Cette configuration est ensuite soumise au module de reconfiguration chargé de détecter les actions à réaliser ainsi qu'un ordonnancement efficace de celles-ci afin d'assurer la fiabilité du processus. Nous discutons de ce module dans le chapitre suivant.

Chapitre 8

La reconfiguration

Où nous discutons du processus assurant la transition entre la configuration courante et une nouvelle configuration viable calculée par un module décision. Cette transition nécessite éventuellement de passer par des configurations transitoires. Nous mettons en évidence l'existence d'inter-dépendances entre certaines actions et proposons un algorithme pour la création d'un plan de reconfiguration assurant la faisabilité de toute les actions de transition tout en favorisant l'exécution en parallèle des actions. Nous évaluons également la durée d'exécution des actions ainsi que leur impact sur les performances dans différentes situations. Ces critères sont en effet prépondérant dans un système auto-adaptatif. Nous définissons alors une approche pour réduire la durée d'une reconfiguration en calculant une configuration impliquant une durée d'exécution du plan associé minimum.

Sommaire

Le passage de la configuration courante à une nouvelle configuration viable implique d'exécuter plusieurs actions agissant sur le cycle de vie des machines virtuelles. Le cycle de vie d'une machine virtuelle est assimilable à une machine à états finis déterministe ; l'observation de la configuration courante et de la configuration destination permet donc d'inférer sur l'ensemble des actions à exécuter. Cette ensemble d'action assurant la transition entre la configuration courante et une nouvelle configuration destination forme une reconfiguration.

Les actions manipulant une machine virtuelle permettent de changer son état ou encore sa position dans la grappe. La Table 8.1 décrit l'impact des actions sur la disponibilité des ressources des nœuds impliqués dans celles-ci à savoir le nœud hébergeant la machine virtuelle dans la configuration courante (le nœud source) et dans la configuration destination (nœud destination).

Les actions de lancement et de reprise d'activité consomment des ressources sur le nœud destination. La disponibilité des ressources sur le nœud destination peut donc être considérée comme une précondition

nécessaire à l'exécution des actions. Les actions d'arrêt et de suspension d'activité au contraire libèrent des ressources sur le nœud source. Cette conséquence est alors une postcondition de l'exécution. Finalement, l'action de migration implique à la fois une précondition sur la disponibilité des ressources sur le nœud destination et d'une postcondition libérant des ressources sur le nœud source.

Action	Impact sur les ressources	
	Consommateur	Libérateur
Lancement	✓	
Arrêt		✓
Suspension		✓
Reprise	✓	
Migration	✓	✓

TABLE 8.1 – Impact des actions sur les ressources des nœuds source et destination

L'existence de préconditions pour l'exécution de certaines actions indique qu'il n'est pas possible de réaliser les actions d'une reconfiguration de manière arbitraire. Si une reconfiguration est composée d'actions libérant et consommant des ressources, alors il peut être nécessaire d'exécuter des actions libérant des ressources en priorité afin de rendre des actions consommatrices réalisables. Il est donc nécessaire de planifier l'ensemble des actions composant un processus de reconfiguration afin d'assurer la faisabilité de chaque action.

Un autre enjeu du processus de reconfiguration est lié à la considération du temps d'exécution des actions. Bien que le temps d'exécution d'une action peut être négligeable dans certaines situations, la réalisation d'un grand nombre d'actions peut nécessiter un temps conséquent lorsque celles-ci sont exécutées séquentiellement. Durant le processus de reconfiguration, les besoins en ressources des machines virtuelles en cours d'exécution sont susceptibles de varier. Lorsque le temps de reconfiguration est long alors l'auto-adaptation ne sera pas assez réactive face à la dynamicité des besoins des machines virtuelles. Le système doit donc assurer une reconfiguration la plus rapide possible pour maximiser la réactivité du système et limiter son impact sur les performances de la grappe.

Nous discutons dans ce chapitre du processus de reconfiguration. Après une description des problèmes liés au séquencement des actions, nous décrivons notre approche assurant la faisabilité des configurations transitoires. Nous discutons de notre approche pour réduire le temps de reconfiguration par la résolution d'un problème d'optimisation sous contraintes. Finalement, nous positionnons notre approche par rapport aux travaux apparentés.

8.1 Détection de dépendances entre actions

Une reconfiguration consiste à exécuter un ensemble d'actions. Les actions de migration, de reprise et de lancement disposent de préconditions relatives à la disponibilité de ressources sur leur nœud d'accueil. Il est donc nécessaire de s'assurer de la disponibilité de ces ressources pour rendre ces actions faisables. Nous discutons dans cette section de notre approche détectant et résolvant différents problèmes de dépendance entre les actions d'une reconfiguration.

8.1.1 Le graphe de reconfiguration

Pour identifier les actions à réaliser lors de la reconfiguration, nous analysons la configuration source et la configuration destination en considérant le cycle de vie d'une machine virtuelle décrit dans le chapitre 6 :

- une machine virtuelle passant de l'état `Attente` à l'état `Exécution` indique une action de lancement ;
- une machine virtuelle changeant de nœud d'accueil mais conservant son état `Exécution` indique une action de migration ;
- une machine virtuelle passant de l'état `Exécution` à l'état `Suspendu`, indique une action de suspension ;

- une machine virtuelle passant de l'état `Suspendu` à l'état `Exécution` indique une action de reprise ;
- finalement, une machine virtuelle n'apparaissant pas dans la configuration destination indique une action d'arrêt.

Un graphe de reconfiguration modélise l'ensemble des actions à réaliser. Il s'agit d'un multigraphe orienté dont les sommets représentent les nœud de calcul et les arcs représentent les actions à réaliser. À chaque sommet sont associées les valeurs de capacité CPU et mémoire du nœud. Sur chaque arc est spécifié le type de l'action ainsi que les besoins en ressources CPU et mémoire de la machine virtuelle associée. La Figure 8.1 représente un graphe de reconfiguration avec 4 actions à réaliser. Nous rappelons que les entiers r_c^j r_m^j définissent respectivement les besoins en ressources CPU et mémoire de la machine virtuelle d'indice j et que les variables c_c^i et c_m^i définissent respectivement la capacité en ressources CPU et mémoire du nœud de calcul d'indice i.

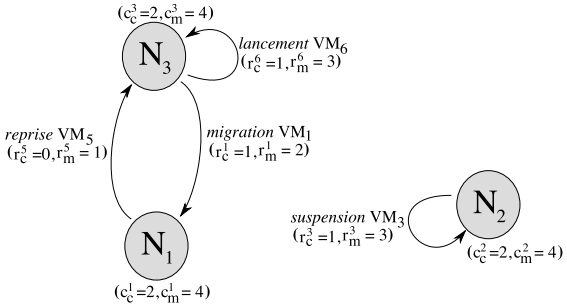


FIGURE 8.1 – Un graphe de reconfiguration

8.1.2 Détection des dépendances

L'analyse des différentes propriétés des actions manipulant les machines virtuelles met en évidence deux types de dépendances. Celles-ci peuvent être repérées et résolues grâce au graphe de reconfiguration.

Dépendance séquentiel. La Figure 8.2 décrit un graphe de reconfiguration précisant la migration de VM_1 du nœud N_1 au nœud N_2 ainsi que la suspension de VM_2 sur son nœud d'accueil N_2. La consommation courante en ressource mémoire de VM_2 réduit la capacité mémoire courante de N_2 qui ne satisfait plus les préconditions nécessaires à l'exécution de la migration de VM_1. Il est donc nécessaire d'exécuter l'action de suspension avant l'action de migration afin de libérer les ressources nécessaires à l'accueil de VM_1.

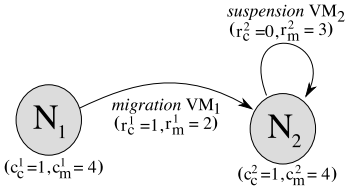


FIGURE 8.2 – Une séquence de deux actions

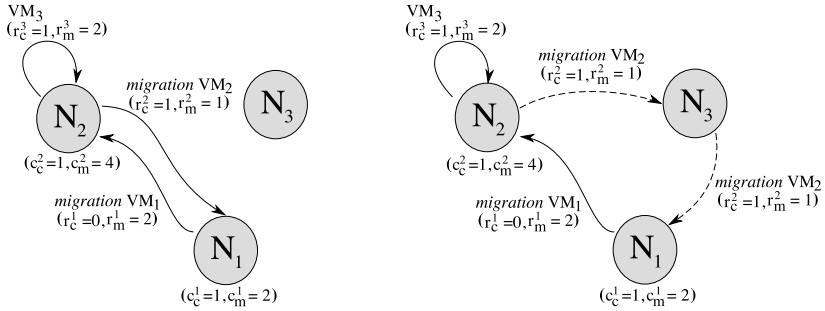
Notons que les actions ne pouvant être exécutées directement sont les actions de migration, de reprise et de lancement. À l'inverse, les actions d'arrêt et de suspension qui ne présentent pas de préconditions, seront nécessairement toujours faisables.

Inter-dépendances. La Figure 8.3(a) représente un graphe de reconfiguration décrivant deux migrations à réaliser. La capacité mémoire du nœud N_1 empêche de réaliser la migration de VM_2 avant la migration de VM_1. Simultanément, la capacité CPU du nœud N_2 empêche de réaliser la migration de

VM_1 avant la migration de VM_2. Ce cycle de dépendances qui consiste à échanger deux ou plusieurs machines virtuelles n'est pas solvable par un séquencement de ces seules actions.

La notion d'échange est observable dans différents domaines. En programmation par exemple, il n'existe pas nécessairement d'opérateur permettant d'échanger les valeurs de deux variables. Pour briser ce cycle de dépendances, il est nécessaire d'utiliser une troisième variable pour stocker la valeur d'une des deux variables. Dans le cas d'un cycle d'actions portant sur des machines virtuelles, nous utilisons un nœud supplémentaire, appelé nœud pivot, pour héberger temporairement l'une des machines virtuelles du cycle. La Figure 8.3(b) décrit une solution pour ce cycle de dépendances. Celui-ci est brisé en migrant d'abord VM_2 sur le nœud pivot N_3 afin de rendre l'action de migration de VM_1 faisable. Finalement VM_2 est migrée vers son nœud destination.

Il est à noter que cette situation ne peut se produire qu'avec des actions de migration. Ce sont en effet les seules actions nécessitant à la fois des ressources sur le nœud destination et libérant des ressources sur le nœud source. De plus, nous devons considérer que la résolution d'un cycle de dépendances par une migration supplémentaire sur un nœud pivot est tributaire de l'existence d'un tel nœud. Une grappe surchargée par exemple ne permettra pas nécessairement de résoudre un problème de cycles par une telle méthode.



(a) Un simple séquencement ne permet pas de migrer VM_1 et VM_2

(b) Une migration supplémentaire sur un nœud pivot brise le cycle de dépendances

FIGURE 8.3 – Inter-dépendance entre deux migrations

La détection et la résolution de ces problèmes de dépendances met en avant le besoin de planifier toutes les actions composant une reconfiguration afin d'assurer la faisabilité de celles-ci.

8.2 Le plan de reconfiguration

Un plan de reconfiguration décrit un ordonnancement d'actions réalisant une reconfiguration qui assure la faisabilité de chacune des actions le composant. Un plan de reconfiguration est donc une solution aux problèmes de dépendances décrit précédemment. Ces dépendances sont résolues en imposant une séquentialisation de certaines actions et éventuellement des migrations supplémentaires. Il n'est cependant pas nécessaire de réaliser la totalité des actions d'une reconfiguration séquentiellement. Un plan de reconfiguration autorise d'exécuter des actions en parallèle, tout en assurant le séquencement des actions disposant de dépendances. Nous avons choisi de modéliser un plan de reconfiguration par une séquence d'étapes. Une étape représente un ensemble d'actions qu'il est possible d'exécuter en parallèle. Une étape peut être composée de séquences d'actions ou d'actions simples. L'enchaînement des différentes étapes est quant à lui séquentiel.

Les actions de lancement, de migration et de reprise possèdent des préconditions relatives à la disponibilité des ressources sur leur nœud destination. Pour décider de la faisabilité de ces actions, nous calculons la capacité libre courante de chaque nœud. Il s'agit de la capacité totale de celui-ci diminuée de la consommation des machines virtuelles en cours d'exécution (cela inclut les machines virtuelles qui vont être migrées, suspendues ou arrêtées). Pour chaque nœud nous construisons une liste d'actions faisable à destination de ce nœud. La liste est initialisée avec l'ensemble des actions prévus à destination du nœud,

triée par ordre croissant selon les besoins mémoire puis CPU des actions. Les actions en queue de liste sont alors progressivement supprimées de la liste jusqu'à ce que la capacité libre courante du nœud soit suffisante pour satisfaire les besoins de toute la liste. L'exécution de cette liste d'actions correspond au pire cas, c'est à dire quand aucune action libératrice de ressources sur le nœud n'est prévue.

Un plan de reconfiguration, initialement vide, est construit itérativement d'après un graphe de reconfiguration entre la configuration source et la configuration destination en calculant à chaque itération la liste des actions faisables de tous les nœuds. Si la liste est vide, alors un cycle de dépendances bloque le processus. Dans cette situation, nous détectons un cycle et un nœud pivot est recherché afin de résoudre ce blocage en créant au moins une nouvelle action faisable. Le pivot est choisi de sorte a exécuter l'action nécessitant le moins de ressources en premier. Si la liste est non vide, alors les actions de celle-ci sont regroupées dans une étape qui est ajoutée en queue du plan de reconfiguration. Nous recréons ensuite un nouveau graphe depuis la configuration transitoire à l'issue de cette étape vers la configuration destination. Ce processus est répété jusqu'à ce qu'il n'y ait plus aucune action dans le graphe.

La Figure 8.4 décrit un exemple de création de plan de reconfiguration. La Figure 8.4(a) décrit le graphe de reconfiguration initiale. À cet instant, les actions *reprise*(VM_4) et *suspension*(VM_5) sont les seules faisables. Ces actions sont ajoutées dans une première étape et nous construisons le nouveau graphe. À ce moment, aucune action n'est faisable. En effet un cycle de dépendances dû aux migrations de VM_1 et VM_2 bloque la situation. Le nœud N_3 est choisi comme pivot afin de réaliser la migration de VM_2 sur N_3 (Figure 8.4(b)), la deuxième étape du plan. Le nouveau graphe représenté par la Figure 8.4(c) indique que la migration de VM_1 est dorénavant faisable ; cette action constitue la troisième étape. Finalement, le nouveau graphe représenté dans la Figure 8.4(d) indique que la migration de VM_2 vers sa destination finale est faisable. Le plan de reconfiguration final, est alors composé de 4 étapes :

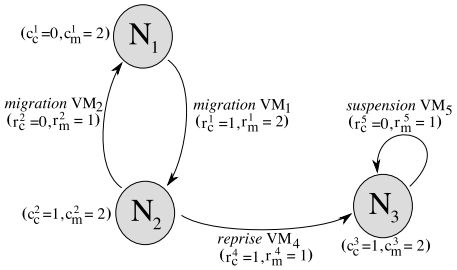
1. suspension(VM_5) et reprise(VM_4)
2. migration(VM_2) sur N_3
3. migration(VM_1)
4. migration(VM_2) sur N_1

8.3 Estimation du coût d'une reconfiguration

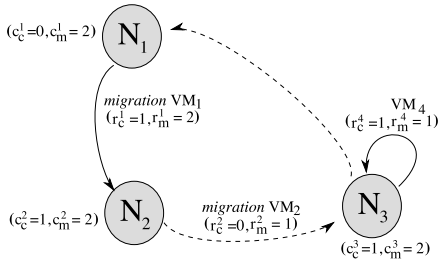
Un processus de reconfiguration consiste à exécuter un ensemble d'actions sur une durée potentiellement longue, dans un environnement dynamique. La durée d'un processus de reconfiguration doit être la plus courte possible pour ne pas impacter les performances de la grappe. Il est donc nécessaire d'estimer la durée des actions afin de pouvoir estimer la durée d'exécution totale d'un plan de reconfiguration. Simultanément, exécuter une action requiert des ressources CPU et des accès à la mémoire. Une action a donc un coût en temps (sa durée d'exécution) mais également en performances qu'il est nécessaire d'évaluer afin d'optimiser au mieux l'exécution d'une reconfiguration.

8.3.1 Méthodologie

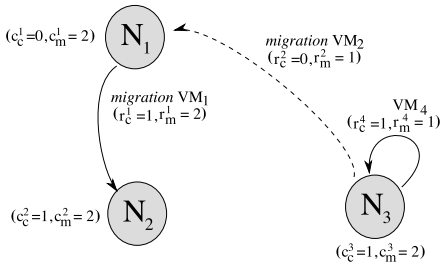
Pour évaluer la durée et l'impact sur les performances des différentes actions, nous avons exécuté celles-ci sur une grappe de test. Ces expériences ont été effectuées sur deux nœuds de calcul identiques disposant chacun d'un processeur Intel Core Duo à 2,1 GHz (un seul cœur est activé) et de 4 Go de mémoire vive. Ces deux nœuds sont interconnectés par un réseau Ethernet Gigabit. L'hyperviseur utilisé est Xen 3.2 et 512 Mo de mémoire sont alloués au Domaine-0. Les machines virtuelles exécutent une distribution GNU/Linux en mode para-virtualisation. La tâche considérée dans ce test est le calcul scientifique BT.W de la suite de banc de test NASGrid [FdW02]. Quand une machine virtuelle exécute le test, elle requiert un CPU physique entier et elle est qualifiée d'« active ». Dans le cas contraire on considère que ses besoins CPU sont nuls, elle est alors qualifiée d'« inactive ». Un nœud est actif s'il héberge des machines virtuelles actives ; dans le cas contraire il est actif. Dans ce contexte, une configuration viable ne peut donc héberger sur un même nœud qu'une seule machine virtuelle active. L'image disque des machines virtuelles est mise



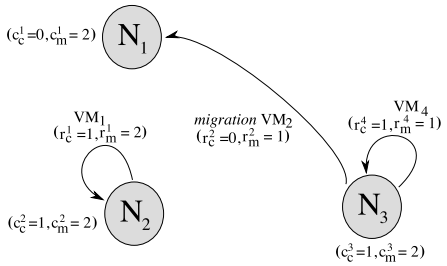
(a) Graphe de reconfiguration initial



(b) Graphe après l'étape 1 et la résolution du cycle de dépendances



(c) Graphe après l'étape 2



(d) Graphe après l'étape 3

FIGURE 8.4 – Processus de création d'un plan de reconfiguration depuis un graphe

Contexte	Migration	Suspension	Reprise
IFITA	✓	✓	✓
IFATI	✓		✓
IFATA	✓		✓
IFITI	✓	✓	✓
AFATI	✓		

TABLE 8.2 – Contextes disponibles pour les actions

à disposition par un serveur NFS. La quantité de mémoire de la machine virtuelle sur laquelle est exécutée l'action varie de 512 Mo à 2048 Mo.

Une action peut être exécutée dans différents contextes dépendant de l'état de la machine virtuelle manipulée ainsi que des nœuds impliqués dans l'action. La Figure 8.5 décrit les différents contextes pour les actions que nous avons définis dans le cadre de notre étude. Le contexte IFITI (*Inactive From Inactive To Active*) manipule une machine virtuelle inactive entre deux nœuds inactifs. Le contexte IFITA (*Inactive From Inactive To Active*) manipule une machine virtuelle inactive entre un nœud inactif et un nœud actif. Le contexte IFATI (*Inactive From Active To Inactive*) manipule une machine virtuelle inactive entre un nœud actif et un nœud inactif. Le contexte IFATA (*Inactive From Active To Active*) manipule une machine virtuelle inactive entre deux nœud actifs. Finalement le contexte AFATI (*Inactive From Active To Active*) manipule une machine virtuelle active entre un nœud actif et un nœud inactif. Nous ne considérons pas le contexte AFATA (*Active From Active To Active*) manipulant une machine virtuelle active vers un nœud en hébergeant déjà une car cette opération entraînerait une configuration non-viable.

La Table 8.2 décrit les contextes qui sont disponibles pour les actions de migration, suspension et reprise. Il est à noter que pour certaines actions le nœud source peut être le nœud destination. Le processus de suspension et de reprise manipule des machines virtuelles qui sont en pause. La machine virtuelle est donc assimilable à une machine virtuelle inactive et dans ce cas, le contexte AFATI manipulant une machine virtuelle active est assimilable au contexte IFATI. Il n'est donc pas nécessaire de traiter ce premier contexte. Dans le cas de la reprise, les contextes IFITA et IFITI peuvent impliquer une reprise sur le nœud hébergeant l'image de la machine virtuelle ou sur un nœud distant. Cependant, bien que la

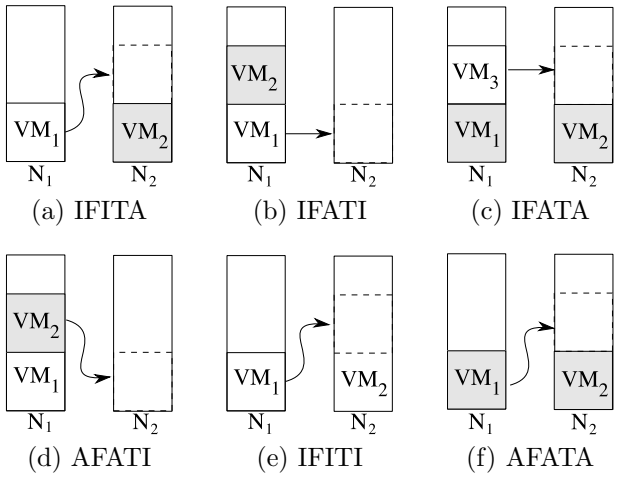


FIGURE 8.5 – Différents contextes pour une action. Les machines virtuelles grises sont considérées comme actives, elles requièrent alors 100% d'un CPU physique

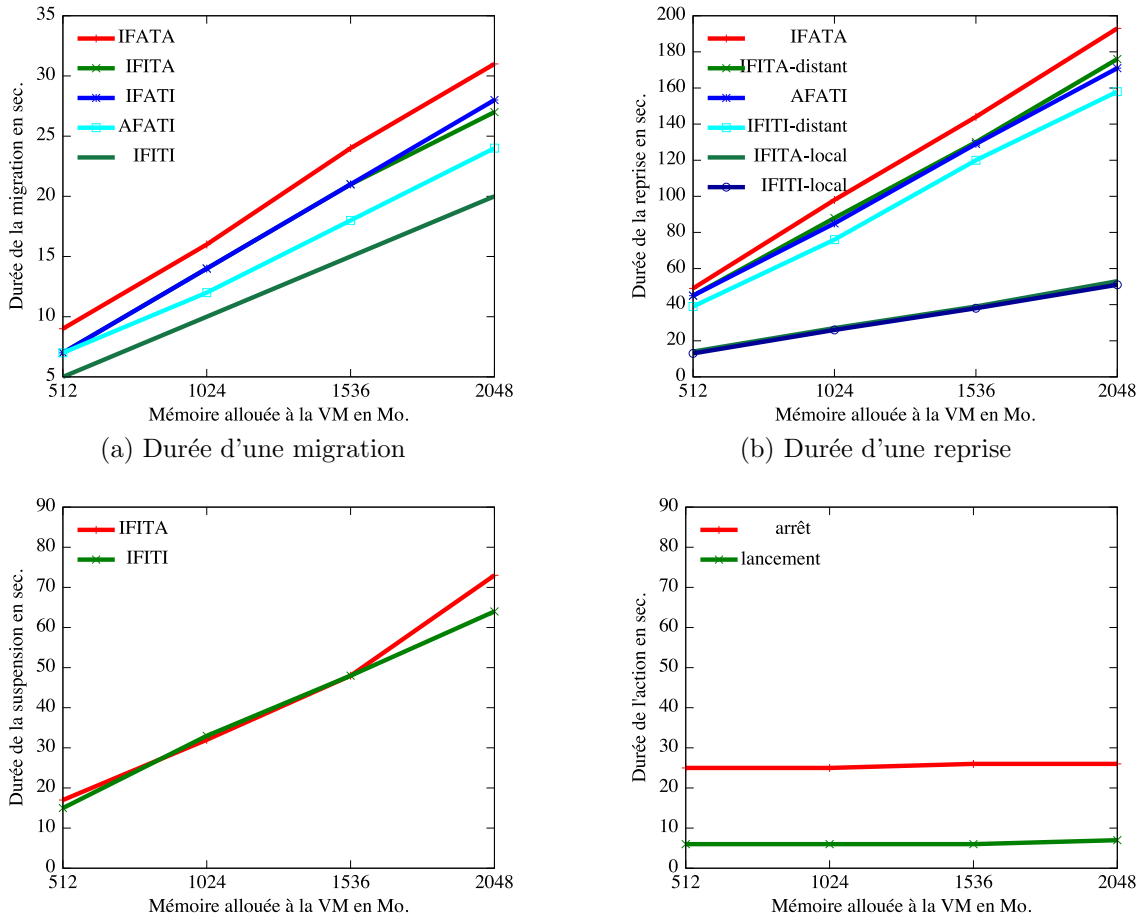
suspension distante soit possible, son intérêt est limité : une machine virtuelle est suspendue dans l'optique d'être reprise ultérieurement, éventuellement sur un nœud de calcul différent. Deux approches sont alors possibles : ou bien suspendre la machine virtuelle sur un nœud de stockage dédié, ou suspendre la machine virtuelle localement afin de la reprendre sur le même nœud ou à distance (en copiant préalablement l'image de la machine virtuelle sur le nœud distant). Dans la première situation, le réseau doit être le plus rapide possible si l'on souhaite obtenir de meilleures performances que dans la seconde situation. En effet, le second cas ne requiert au pire qu'un transfert d'image par le réseau. L'architecture nécessaire pour valoriser l'approche de la suspension sur un nœud dédié étant peu commune, nous ne considérons pas, dans cette étude, de contextes pouvant impliquer une suspension sur un nœud distant.

8.3.2 Durée des actions

La Figure 8.6 référence la durée moyenne d'exécution des différentes actions manipulant les machines virtuelles en fonction des contextes définis précédemment et de la quantité de la mémoire allouée à la machine virtuelle manipulée. On observe que le temps de migration, de reprise et de suspension (visibles respectivement sur les Figures 8.6(a), 8.6(b) et 8.6(c)) est guidé principalement par la quantité de mémoire allouée à la machine virtuelle. En effet, ces actions manipulent surtout la mémoire de la machine virtuelle, il est donc naturel que leur durée dépende de cette quantité. Le contexte a un impact fondamental dans le cadre d'actions de reprise. Les contextes impliquant une reprise à distance prennent naturellement plus de temps que les contextes où la reprise est locale. Il est en effet nécessaire de transférer l'image de la machine virtuelle sur le nœud distant[1]. On observe alors que le temps d'exécution d'une action est multiplié par 3. L'état des nœuds impact également sur le temps de réalisation de l'action. La gestion de la copie et de la restauration de l'image nécessite des ressources CPU qui sont alors prélevées sur les besoins des machines virtuelles actives.

La Figure 8.6(d) montre que le temps d'exécution des actions de lancement et d'arrêt est indépendant de la quantité de mémoire allouée à la machine virtuelle manipulée. Dans la pratique, le temps de lancement et d'arrêt est en effet principalement consacré au lancement des différents services du système d'exploitation de la machine virtuelle. Comparé au temps de démarrage (environ 6 secondes), le temps d'arrêt d'une machine virtuelle est important (environ 25 secondes), il est cependant possible de le réduire à moins d'une seconde en arrêtant la machine virtuelle brutalement (équivalent à l'arrêt de l'alimentation d'une machine physique). Le gain en temps est significatif par contre l'utilisateur devra considérer cette particularité afin d'adapter si nécessaire son environnement.

[1]Dans cette étude, l'image est copiée grâce à la commande `scp` bien que différentes approches soient possibles (`rsync`, serveur NFS, ...).



(a) Durée d'une migration

(b) Durée d'une reprise

(c) Durée d'une suspension

(d) Durée des actions de lancement et d'arrêt

FIGURE 8.6 – Durée d'exécution des actions manipulant une machine virtuelle

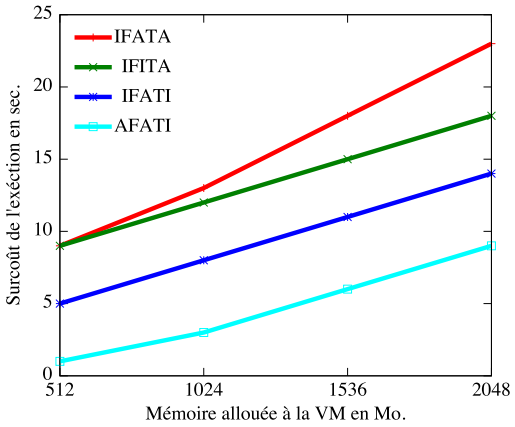
8.3.3 Impact sur les performances

Les Figure 8.7 représentent le surcoût de l'exécution d'un test selon les différentes actions et contextes par rapport à une exécution sans action. Pour cette évaluation, nous ignorons les contextes n'impliquant que des machines virtuelles inactives, celles-ci n'ayant aucun besoins CPU, on ne peut observer de pertes.

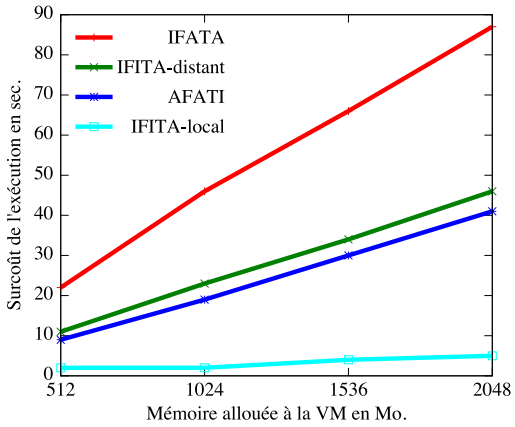
L'impact sur les performances dans le cadre d'une migration (voir Figure 8.7(a)) est dépendant du contexte de l'action. Pour le contexte IFATI, le surcoût provient de la lecture des pages mémoire sur le nœud source. En effet, l'écriture des pages sur le nœud destination n'a pas d'impact puisqu'il n'héberge aucune machine virtuelle active. Pour le contexte AFATI, la migration porte sur une machine virtuelle active. Le surcoût est donc légèrement supérieur puisqu'un nombre de passes supérieur est nécessaire pour copier le contenu de la mémoire qui change fréquemment et que les ressources nécessaire à la lecture de ces pages réduit la quantité de ressources CPU accordée à la machine virtuelle. Le contexte IFITA est soumis à un surcoût encore supérieur puisque le processus d'écriture des pages mémoire sur le nœud destination requiert des ressources CPU qui ne sont alors plus disponibles pour VM_2 qui est active. Finalement, le contexte IFATA est le contexte qui implique le surcoût le plus important. En effet, même si la machine virtuelle migrée n'est pas active, le processus de lecture de pages sur N_1 et d'écriture sur N_2 prive les machines virtuelles actives de ressources CPU qui étaient demandées. Ce surcoût est comparable à la somme des surcoûts des contextes IFATI et IFITA.

La Figure 8.7(b) représente le surcoût de l'action reprise dans différents contextes. Encore une fois, les contextes impliquant des reprises à distance sont nettement plus coûteux en performance que les actions locales (contexte IFITA-local). La copie de l'image sur le nœud distant demandant des ressources CPU

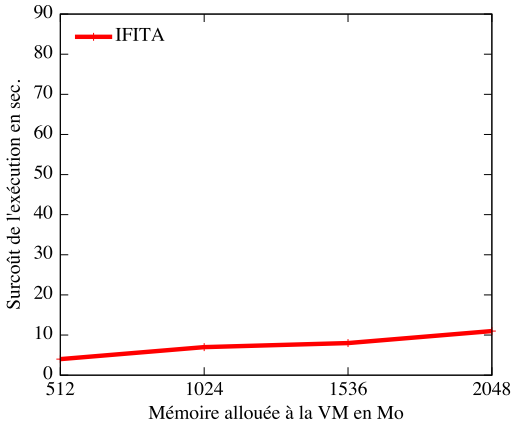
sur les nœuds source et destination, nous observons que la copie d'une image est plus impactante sur le nœud destination lorsqu'il est actif (contexte IFITA-distant) que sur le nœud source (contexte AFATI) ; l'écriture de données sur disque étant plus coûteuse en ressources CPU que la lecture. La Figure 8.7(c) décrit le surcoût d'une suspension locale sur un nœud actif.



(a) Impact d'une migration



(b) Impact d'une reprise



(c) Impact d'une suspension

FIGURE 8.7 – Impact des actions manipulant une machine virtuelle sur les performances

Globalement, nous observons que l'exécution d'une action sur une machine virtuelle active ou sur des nœuds actifs dégradent significativement les performances des machines virtuelles présentes sur le nœud source ou destination. Nous avons observé que les opérations les moins coûteuses manipulent des machines virtuelles inactives entre des nœuds inactifs, et que les contextes où le nœud destination est actif dégradent plus les performances que les contextes ou le nœud source est actif. De plus, il est important de noter que même s'il est possible de déplacer une machine virtuelle en fonctionnement par l'intermédiaire d'une suspension et d'une reprise à distance, il est impératif d'utiliser une migration à la place. En effet, pour un même résultat, la durée de son exécution est 9 fois inférieure et l'impact sur les performances des machines virtuelles actives est 7 fois moindre. Ces expérimentations suggèrent de réduire au minimum le nombre d'actions à réaliser, en évitant les migrations par exemple. Elles suggèrent également de considérer le contexte de l'action, en préférant la suspension et la reprise locale par exemple, tout en manipulant le plus possible des machines virtuelles et des nœuds inactifs.

8.4 Réduction de la durée de la reconfiguration

L'évaluation de la durée des actions dans la section précédente a mis en avant que certains contextes de réalisation d'actions étaient préférables à d'autres. En s'appuyant sur notre modèle de plan défini précédemment, nous avons défini une fonction objectif estimant le coût, en temps, d'un plan de reconfiguration. Cette fonction peut ainsi être utilisée dans un modèle d'optimisation sous contraintes afin de calculer une configuration, équivalente à la configuration destination calculée par le module de décision, mais dont le coût du plan de reconfiguration associé est minimisé.

8.4.1 Modèle de coût

Le coût d'un plan de reconfiguration p, composée d'une séquence de n étapes e et de m actions a est défini au travers d'une fonction objectif K de l'équation (8.1). Le coût d'un plan de reconfiguration correspond à la somme de tout les coûts totaux $K(a_x)$ de chaque action. Le coût total d'une action correspond à son coût local $k(a_x)$ plus le coût des différentes étapes précédents l'étape où l'action est exécutée. Comme nous exécutons les actions d'une étape en parallèle, le coût d'une étape correspond au coût local son action la plus coûteuse. Le coût local k d'une action a_x est défini d'après les expériences décrites dans la section précédente. Ces coûts sont résumés dans la Table 8.3. Nous avons montré que le coût des actions de lancement et d'arrêt était constant. Dans le contexte de cette fonction, ce coût a été ramené à 0. La durée d'une action de migration ou de suspension est proportionnelle à la quantité de mémoire manipulée. Dans le contexte de cette fonction, ce coût est égal à cette quantité. Finalement, la durée d'une action de reprise est guidée par la quantité de mémoire, mais également par son contexte (local ou distant). Ainsi, expérimentalement, nous considérons que le coût d'une reprise distante est le double du coût d'une reprise locale.

$$
\begin{aligned}
K\left(p\right) &= \sum_{x=0}^{m} K\left(a_x\right) \\
a_x \in s_i, K\left(a_x\right) &= \sum_{j=0}^{i-1} K\left(s_j\right) + k\left(a_x\right) \\
K\left(s_i\right) &= max\left(k\left(a_x\right)\right), a_x \in s_i
\end{aligned}
\tag{8.1}
$$

Type de l'action a_x	Coût local $k\left(a_x\right)$
migration(VM_j)	r_m^j
lancement(VM_j)	*constant*
arrêt(VM_j)	*constant*
suspension(VM_j)	r_m^j
reprise(VM_j)	r_m^j si *local*, $2 \times r_m^j$ sinon

TABLE 8.3 – Coût des différents types d'actions sur une machine virtuelle d'indice j. r_m^j indique la quantité de mémoire allouée à VM_j.

Dans la pratique, minimiser cette fonction de coût :

- favorise le localisme en pénalisant les reprises distantes par rapport aux reprises locales et les migrations non nécessaire ;
- réalise les actions au plus tôt en favorisant le parallélisme.

  Une reconfiguration a lieu lorsque le module de décision détecte une configuration non viable ou détecte qu'une meilleure configuration est possible. Si la configuration courante est non viable, alors certaines machines virtuelles en cours d'exécution n'ont pas accès à suffisamment de ressources pour une exécution optimale. En exécutant les actions le plus tôt possible, c'est à dire en réduisant le

coût total d'une action, nous minimisons le temps où les performances des machines virtuelles sont réduites ;

- choisit des migrations de machines virtuelles demandant peu de mémoire afin de réaliser l'opération le plus rapidement possible.

8.4.2 Minimisation du coût d'un plan

Un problème d'affectation de machines virtuelles comme VMPP, qui calcule une configuration viable utilisant le minimum de nœuds, peut avoir plusieurs solutions. Celles-ci ont en commun l'état de chacune des machines virtuelles, mais elles différent par le placement des machines virtuelles. On considère que deux configurations sont équivalentes si elles impliquent les mêmes machines virtuelles et que celles-ci sont toutes dans le même état dans chaque configuration. On préférera alors une configuration où le coût du plan de reconfiguration associé est le plus petit.

Le problème du calcul d'une configuration viable impliquant le coût de reconfiguration le plus faible est appelé VMRP (*Virtual Machines Replacement Problem*). Ce problème d'optimisation sous contraintes est calculé dans le module de planification à partir de la solution calculée par le module de décision. Il utilise la fonction de coût K définie précédemment et le modèle du VMAP défini dans le chapitre 6. Nous avons composé ce problème en rajoutant les contraintes de sac à dos afin de calculer des configurations viables et les différentes contraintes qui maintiennent les états des machines virtuelles (la définition de ces contraintes est rappelée en annexe). Nous réutilisons également les heuristiques de recherche permettant une approche *first fail*.

Afin de réduire l'espace de recherche de ce problème, nous calculons les bornes de la variable objectif K. La borne supérieure correspond au coût du plan entre la configuration courante et la configuration calculée par le module de décision. La borne inférieure quant à elle calculée à partir des actions qui seront nécessairement exécutées, c'est à dire les actions de lancement, d'arrêt, de suspension et de reprise. Pour ces actions, le coût minimum est choisi.

Si le module de décision calcule une configuration ayant des propriétés d'équivalence autres que l'état des machines virtuelles, alors il est nécessaire de préciser au module de planification les contraintes supplémentaires à considérer. Ainsi, si le module de décision utilise VMPP, alors VMRP doit s'assurer que la solution utilisera le même nombre de nœud que le nombre x_{vmpp} calculé par VMPP. La contrainte définie par l'équation (8.2) est alors ajoutée au problème.

$$\sum_{i \in \mathcal{N}} u_i = x_{vmpp} \tag{8.2}$$

Tout comme pour le VMPP, il est possible de considérer les équivalences entre les machines virtuelles et les nœuds afin de réduire l'espace de recherche. Dans cette situation cependant, la relation d'équivalence entre les machines virtuelles doit être plus restrictive. En effet, cette relation d'équivalence doit considérer que les machines virtuelles peuvent se déplacer sur un autre nœud. Ainsi, deux machines virtuelles sont équivalentes si leurs besoins en ressources CPU et mémoire sont identiques et si elles sont hébergées par le même nœud. Cette nouvelle relation d'équivalence est formalisée dans l'équation (8.3).

$$\forall x, y \in [0, k], \mathrm{VM}_x \equiv \mathrm{VM}_y \Leftrightarrow r_c^x = r_c^y \wedge r_m^x = r_m^y \wedge v_x = v_y \tag{8.3}$$

Nous avons également développé une contrainte accélérant la minimisation de la fonction objectif. Lors du test d'une affectation d'une machine virtuelle à un nœud, un plan de reconfiguration partiel est calculé et son coût est évalué. Ce coût est ensuite complété en estimant un coût minimum pour les machines virtuelles non affectées (nous reprenons le calcul de la borne inférieure de K). Si ce coût est supérieur à la borne supérieure alors le solveur teste une autre affectation. Si le coût est inférieur à la borne supérieure et que le solveur a calculé une solution, alors ce coût représente une nouvelle borne supérieure.

8.5 Travaux apparentés

Différents travaux ont abordé la notion de reconfiguration, dans le cadre de serveur web pour l'équilibrage de charge, de la migration de données dans les grappes, ou de la consolidation de machines virtuelles. Bien que la finalité de ces approches soit différente, toutes consistent à calculer un plan assurant la transition entre une configuration courante et une configuration destination.

Dans le domaine de l'équilibrage de charges, Aggarwal *et al.* [AMZ03] proposent des algorithmes d'optimisations approchés permettant de passer d'une configuration à une autre en minimisant le nombre de déplacements des tâches. Récemment, Fukunaga [Con08] a proposé une approche exacte reposant sur la programmation par contraintes pour la résolution d'un problème équivalent.

Hall *et al.* [HHK+01] et Anderson *et al.* [AHH+08] proposent une série d'algorithmes pour la création de plans de migration de données dans les grappes. L'algorithme proposé en 2008 considère des problèmes de dépendances évoqués dans cette thèse mais se basent sur des hypothèses invalides dans notre contexte de migration de machines virtuelles : les données à migrer ont toute la même taille et chaque nœud de la configuration initiale et finale doit disposer de suffisamment d'espace libre pour accueillir les données. Dans le cadre de la migration de machines virtuelles, la quantité de mémoire allouée à celles-ci est différente et les nœuds ne disposent pas nécessairement d'espace libre pour accueillir des machines virtuelles supplémentaires. Notons cependant que notre algorithme de création de plan ne nécessite au plus qu'un nœud avec suffisamment de ressources CPU et mémoire libres pour accueillir la machine virtuelle la plus consommatrice en ressources.

Différents travaux récents ont également adressé le problème de la reconfiguration dans les grappes exécutant des machines virtuelles. Verma *et al.* [VAN08b] proposent un algorithme pour concentrer les machines virtuelles sur un minimum de nœuds. En ne considérant uniquement que des migrations directement faisables, il n'est pas nécessaire de planifier ces actions. Cette approche limite cependant fortement les possibilités de reconfiguration en ne considérant qu'un sous ensemble de solutions possibles. Cette considération dégrade la qualité des solutions par rapport à une approche exacte : dans le cas de grappes soumises à une forte charge par exemple, cette approche ne parviendra pas nécessairement à déplacer les machines virtuelles permettant de retrouver une configuration viable.

Wood *et al.* [WTLS+09] considèrent les dépendances séquentielles dans leur environnement de consolidation de machines virtuelles mais pas les cycles de dépendances. Grit *et al.* [GIYC06] considèrent dans Shirako [ICG+06] les dépendances séquentielles et cycliques. Cependant les cycles sont tous résolus en remplaçant une migration par une suspension et une reprise de la machine virtuelle. Notre approche résout les problèmes de cycles en insérant une migration supplémentaire vers un nœud pivot. Cette solution permet de traiter des applications à haute disponibilité dans les machines virtuelles et dispose d'un coût de réalisation et un impact sur les performances nettement inférieur.

Globalement, tous ces travaux proposent des algorithmes réutilisables pour la création de plans de reconfiguration. Ces approches résolvent seulement une partie des problèmes de dépendances évoqués dans ce chapitre ou bien contournent le problème en limitant les actions réalisables. De plus, aucune de ces approches ne cherche à remettre en cause la configuration destination afin d'optimiser le plan de reconfiguration ou ne proposent une étude poussée des coûts des actions. Nous proposons dans cette thèse, en plus d'un algorithme de planification résolvant les dépendances séquentielles et cycliques, une approche exacte pour le calcul d'une configuration équivalente à celle calculée par le module de décision mais impliquant un coût de reconfiguration minimum.

8.6 Conclusion

Bilan

Nous avons présenté dans ce chapitre une approche dédiée à la préparation de la transition entre la configuration courante et une configuration destination viable. Un tel processus implique de réaliser différentes actions manipulant l'état et la position des machines virtuelles dans la grappe. L'étude des différentes actions a montré qu'une planification était nécessaire afin de résoudre les problèmes de dépendances séquentielles et cycliques. Tandis que l'étude des coûts des actions selon leur contexte a révélé l'intérêt de minimiser la durée de la reconfiguration. Nous avons proposé un modèle de plan de reconfiguration et

un algorithme ordonnant toutes les actions de manière à assurer leur faisabilité tout en maximisant le parallélisme. De par l'étude des différents contextes agissant sur la durée d'une action et sur la perte de performances associée, nous avons modélisé une fonction objectif estimant le coût (en temps) d'un plan de reconfiguration. Cette fonction a pu ensuite être incluse dans un problème d'optimisation sous contraintes, dérivé du VMAP, afin de calculer une configuration équivalente à la configuration obtenue auprès du module de décision, mais dont le plan associé a le coût le plus petit possible. Nous pensons que cette optimisation permettra de réduire le temps de reconfiguration, critère clef assurant la réactivité de l'auto-optimisation de l'agencement des machines virtuelles dans Entropy. La validité de cette hypothèse sera vérifiée expérimentalement dans le Chapitre 10.

Perspectives

Différentes améliorations sont encore envisageables. Premièrement, notre algorithme de création de plan nécessite qu'un nœud de la grappe dispose d'un espace libre suffisamment grand pour servir de pivot en cas de dépendance cyclique. Si cette situation est envisageable dans la majorité des cas, il est intéressant de proposer une alternative reposant sur la suspension et de reprise d'une machine virtuelle. Dans ce cas, lorsqu'un cycle de dépendances n'est pas résolvable, alors nous réalisons une suspension d'un machine virtuelle puis sa reprise lorsque les dépendances ont été résolues. Cette solution assure la faisabilité d'une reconfiguration dans toute les situations, sans pour autant utiliser par défaut la préemption d'une machine virtuelle pour résoudre un cycle, méthode beaucoup plus coûteuse en temps et en performances. Deuxièmement, l'étude des contextes des actions, notamment l'impact sur les performances montre un intérêt à considérer celui-ci afin d'obtenir un plan de reconfiguration minimisant la perte de performances. Il faudrait alors confronter ces deux approches pour l'optimisation de la reconfiguration et sélectionner la solution la plus profitable où définir une fonction de coût réalisant un compromis. Finalement, le calcul de la borne inférieure du coût de la reconfiguration ne tient pas compte des problèmes de planification. La valeur obtenue est donc relativement imprécise. En développant des heuristiques plus avancées pour l'estimation d'un coût minimum pour une reconfiguration, nous pourrions réduire le temps de calcul de notre approche.

La séparation entre le module de décision et le module de planification assure le maintien des états des machines virtuelles ainsi que la viabilité de la configuration. Le module de planification peut cependant calculer une nouvelle configuration disposant d'un agencement des machines virtuelles différents. Si l'algorithme d'ordonnancement contraint le placement des machines virtuelles, ces contraintes ne seront pas prisent en compte. Il est donc nécessaire de pouvoir spécifier des contraintes de placement qui seront satisfaites par le module de planification. Ce besoin a été exposé lorsque nous demandons au module de planification de calculer une configuration utilisant un nombre de nœuds précis pour l'hébergement. Il est donc nécessaire de pouvoir spécifier en plus d'une configuration exemple, les contraintes de placement que le module de planification devra respecter. Il convient alors de revoir le modèle de communication entre le module de décision et le module de planification afin de préciser au choix une relation d'équivalence qui doit être satisfaite entre la configuration exemple et la configuration destination ou un ensemble de contraintes additionnelles passées au module de planification.

Chapitre 9

Ordonnancement flexible de tâches

Où nous discutons de la généralisation du problème de gestion du placement des machines virtuelles en proposant une architecture permettant aux administrateurs de développer leur propre stratégie d'ordonnancement, sélectionnant les tâches à exécuter sur la grappe. En se basant sur la séparation des concepts de décision et les mécanismes assurant la transition, notre approche permet aux développeurs de ne pas avoir à considérer les problèmes liés à la reconfiguration. Le développement de stratégies d'ordonnancement complexe se voit alors simplifié.

Sommaire

Nous avons discuté dans le chapitre 7 d'un premier module de décision pour la consolidation dynamique. Ce module calcule une configuration viable où les tâches en cours d'exécution sont hébergées sur un nombre de nœuds minimum, dans le but d'augmenter la capacité d'accueil de la grappe ou de réduire les coûts de fonctionnement de la grappe en éteignant les nœuds inutilisés. Nous avons cependant expliqué que cette solution ne peut résoudre les situations où la grappe est surchargée et qu'une solution possible consiste à manipuler l'état des machines virtuelles en plus de leur placement. Nous avons décrit dans le chapitre 2 que les stratégies d'ordonnancement réalisant une gestion dynamique des tâches nécessitaient de manipuler l'état des tâches et de leur composants. La stratégie d'ordonnancement *gang scheduling* avec concentration des tâches par exemple, requiert de migrer les machines virtuelles des tâches pour les exécuter sur un minimum de quantums, tandis que la transition entre les quantums est réalisée par des actions de suspension et de reprise sur les machines virtuelles.

Nous avons défini avec Entropy une architecture où la manipulation des machines virtuelles est réalisée en deux étapes. D'abord le module de décision, calcule une configuration viable indiquant l'état des machines virtuelles pour la prochaine itération puis le module de reconfiguration planifie la transition entre la configuration courante et la nouvelle configuration en réduisant le coût de la reconfiguration au minimum. Il calcule pour cela une configuration équivalente à la configuration initialement calculée, où les états des machines virtuelles seront conservés mais où l'agencement des actions et leur nombre sont optimisés pour réduire le temps d'exécution du plan. Ce processus de transition entre deux états de la grappe, symbolisés par des configurations, peut être assimilé à un changement de contexte tel que celui réalisé dans les systèmes d'exploitation exécutant des processus en pseudo-parallèle : un ordonnanceur décide quels processus seront exécutés sur des processeurs puis suspend les processus qui devront passer en attente afin d'exécuter les processus choisis.

Nous présentons dans ce chapitre notre approche pour l'ordonnancement de tâches composées de machines virtuelles et le principe de changement de contexte dans les grappes appliqué aux tâches composées

de machines virtuelles. Le changement de contexte dans Entropy consiste à exécuter une transition entre deux configurations qui peut impliquer la migration de machines virtuelles en cours d'exécution, mais également le démarrage, la suspension, la reprise et l'arrêt des machines virtuelles d'une tâche. Ce concept reprend le principe de la reconfiguration décrit dans le Chapitre 8 en considérant cette fois des actions sur des tâches composées de plusieurs machines virtuelles. Cette approche permet aux administrateurs de développer des stratégies d'ordonnancement pour grappes complexes en indiquant à un module de décision dédié à l'ordonnancement des tâches à exécuter pour la prochaine itération de la boucle de contrôle. Le module de planification assure alors le changement de contexte le plus rapidement possible.

Nous présentons dans une première section de ce chapitre le principe général de notre approche et son implémentation dans Entropy puis dans une deuxième section, nous présentons un premier algorithme d'ordonnancement exécutant les tâches d'une file d'attente au plus tôt. Nous discutons dans une troisième partie des travaux apparentés. Finalement, nous concluons ce chapitre en présentant les limites de notre approche et les extensions possibles.

9.1 Architecture

Réaliser un ordonnanceur de tâche flexible consiste d'abord au développement d'un algorithme d'ordonnancement. Cet algorithme calcule un nouvel état pour toutes les tâches d'après l'observation de la configuration courante de la grappe. Le résultat de cet algorithme est ensuite utilisé par un module de décision dédié à l'ordonnancement pour calculer une nouvelle configuration viable. Le changement de contexte sera alors préparé et optimisé par le module de planification en observant la configuration courante et la nouvelle configuration.

9.1.1 Un module de décision dédié à l'ordonnancement de tâches

Un module de décision pour l'ordonnancement est dédié au calcul d'une configuration viable indiquant l'état des machines virtuelles pour la prochaine itération. Ce module est personnalisé par l'administrateur qui peut accéder aux données de supervision et utiliser ses propres algorithmes d'ordonnancement pour sélectionner les tâches à exécuter. Il peut par exemple définir plusieurs listes d'attentes et différents critères de sélection comme l'heure de soumission des tâches ou les besoins courants des machines virtuelles composant ses tâches. L'algorithme d'ordonnancement est ensuite utilisé par le module de décision dédié à l'ordonnancement. Celui-ci a alors la charge d'optimiser la nouvelle configuration qui sera ensuite utilisée pour planifier le changement de contexte.

Nous définissons par RJAP (*Running Job Assignment Problem*) le problème du calcul d'une nouvelle configuration viable d'après l'algorithme d'ordonnancement. Ce problème de satisfaction de contraintes est modélisé en se basant sur le problème basique VMAP définis dans le Chapitre 6. Il assure que chaque machine virtuelle de la configuration résultat sera dans l'état indiqué par l'algorithme d'ordonnancement. La modélisation de RJAP inclue également les contraintes de sac à dos pour assurer la viabilité de la configuration ainsi que la contrainte de filtrage additionnel utilisant les nœuds et machines virtuelles équivalentes (la définition de chacune de ces contraintes est disponible en annexe). Les heuristiques de recherche définies dans le Chapitre 7 sont également implémentées pour accélérer le processus de résolution.

Le Listing 9.1 décrit l'implémentation de ce module de décision. À la ligne 6 et 7, nous instantions l'algorithme d'ordonnancement et récupérons son résultat. À partir de la ligne 9, nous utilisons l'API d'Entropy pour modéliser et résoudre RJAP : de la ligne 11 à 26, nous nous assurons de maintenir l'état des machines virtuelles. Nous ajoutons ensuite les contraintes de sac à dos. Finalement, nous considérons les symétries dans l'affectation pour accélérer le processus de résolution (ligne 37) et nous spécifions les heuristiques de recherche `StayFirstNodeSelector` pour réduire le nombre de migrations et de reprises distante.

```
public class RJAP extends DecisionModule {
  @Override
  public Configuration compute() throws Exception {
      Configuration current = this.getCurrentObservation().getConfiguration();
```

```
5
        DecisionModule dm = ...;
        Configuration sample = dm.compute();

        ChocoSolver model = new ChocoSolver(current);
10
        for (VirtualMachine vm : sample.getRunnings()) {
      model.mustBeRunning(vm);
        }
        for (VirtualMachine vm : sample.getWaitings()) {
15    model.mustBeOnFarm(vm);
        }
        for (VirtualMachine vm : sample.getSleepings()) {
      model.mustBeOnFarm(vm);
        }
20      for (VirtualMachine vm : model.getInitialConfiguration().getRunnings()) {
      if (!sample.getRunnings().contains(vm) &&
          !sample.getWaitings().contains(vm) &&
          !sample.getSleepings().contains(vm)) {
          model.mustBeStopped(vm);
25    }
        }

        for (Node node : sample.getOnlines()) {
      model.setKnapSackOnNode(node,
30            Node.MEMORY_TOTAL,
              VirtualMachine.MEMORY);
      model.setKnapSackOnNode(node,
              Node.CPU_CAPACITY,
              VirtualMachine.CPU_CONSUMPTION);
35      }
        if (sample.getRunnings().size() > 0) {
      model.addConstraint(new SymetryBreakingVMPP(model));
        }
        VirtualMachineSelector s = new BiggestVirtualMachinesFirst(model,
40                    VirtualMachine.MEMORY);
        model.setVirtualMachineSelector(s);
        model.setNodeSelector(new StayFirstNodeSelector(model));
        if (model.solve(this.getTimeout())) {
      return model.getResultingConfiguration();
45      }
        return model.getResultingConfiguration();
    }
}
```

Listing 9.1 – Extrait de l'implémentation de RJAP dans Entropy

9.1.2 Implémentation du changement de contexte de tâches

Réaliser un changement de contexte de tâches à base de machines virtuelles implique de changer l'état d'un ensemble de machines virtuelles appartenant à une même tâche de manière cohérente. Lorsqu'une tâche exécute une application distribuée, chaque composant de celle-ci est embarqué dans une machine virtuelle et communique avec les autres au travers d'un réseau par un protocole tel que TCP. Ce dernier tolère la perte de paquets et leur déséquencement et autorise un délai maximum d'attente avant de considérer la connexion perdue.

Nous avons considéré dans le Chapitre 6, lors de la définition d'une configuration, que toutes les machines virtuelles d'une même tâche devaient être dans le même état. Cependant, il n'existe pas d'actions

permettant de changer l'état de l'ensemble des machines virtuelles d'une tâche de manière atomique : les actions de lancement, d'arrêt, de suspension et de reprise permettent de changer uniquement l'état d'une seule machine virtuelle. Durant la reconfiguration, les configurations transitoires résultantes de l'application d'une étape peuvent donc décrire des tâches où les machines virtuelles seront potentiellement dans des états différents pendant un temps supérieur à celui toléré par le protocole de transport. Cette situation met les applications distribuées en faute. Si l'on souhaite empêcher ces erreurs, nous devons nous assurer que la suspension et la reprise des machines virtuelles d'une même tâche n'entraînent pas une perte de paquets non absorbable par le protocole de transport, mais également que le délai maximum entre l'exécution de la première et de la dernière action est inférieur au délai maximal toléré par le protocole.

Différentes expériences tel que celles conduites par Emeneker *et al.* [ES06] montrent que les applications distribuées ne sont pas mises en faute lorsque les actions de suspension et la reprise des machines virtuelles sont réalisées toujours dans le même ordre et au même moment ou dans un intervalle de temps court. L'algorithme de création de plan décrit dans le Chapitre 8 ne permet pas de réunir ces conditions car il considère chaque machine virtuelle indépendamment.

Nous proposons ainsi de modifier le plan de reconfiguration pour regrouper les actions manipulant une même tâche afin de toutes les exécuter dans une courte période. Les actions de suspension de tâches sont naturellement regroupées dans la première étape (elles sont par principe toujours faisables). Les actions de reprise sont quant à elles regroupées dans la dernière étape contenant une de ces actions. Il n'est en effet pas possible de les regrouper dans une étape antérieure sans se heurter à des problèmes de dépendances entre actions. Afin de maintenir l'ordre d'exécution des actions, celles-ci sont triées selon l'identifiant de la machine virtuelle. Pour assurer un délai d'exécution des actions inférieur au délai maximal d'attente du protocole de transport, les actions de suspension et de reprise sont toutes décomposées en deux sous-actions. Dans le cas de la suspension, nous réalisons d'abord une pause de la machine virtuelle, cette opération, d'une durée très courte, suspend l'activité de la machine virtuelle mais laisse son état dans la mémoire vive. Ensuite, la deuxième sous-action sauvegarde l'état de la machine virtuelle dans un fichier. L'action de reprise réalise ces deux actions dans le sens inverse. La suspension des machines virtuelles d'une tâche consiste alors à exécuter séquentiellement toutes les actions de pause, puis d'exécuter en parallèle toutes les actions de sauvegarde d'état. La reprise quant à elle réalise d'abord une restauration de toutes les machines virtuelles en parallèle puis relance toutes les machines virtuelles séquentiellement. Ce processus assure la fiabilité de l'opération, tout en exécutant en parallèle les opérations d'écriture et de lecture de la mémoire qui sont coûteuses en temps et en performance.

Les Figures 9.1 et 9.2 décrivent respectivement un plan de reconfiguration basique et sa version modifiée qui maintient la cohérence des états de 3 tâches j_1, j_2 et j_3. La tâche j_1 est composée des machines virtuelles vm_1, vm_2 et vm_3, la tâche j_2 est composée des machines virtuelles vm_4 et vm_5 et la tâche j_3 est composée des machines virtuelles vm_6, vm_7 et vm_8.

s_1 : $suspension(vm_1)$ & $suspension(vm_2)$ & $suspension(vm_3)$ & $migration(vm_6)$
s_2 : $reprise(vm_4)$
s_3 : $migration(vm_7)$
s_4 : $reprise(vm_5)$ & $migration(vm_8)$

FIGURE 9.1 – Plan de reconfiguration initiale en 4 étapes exécutées séquentiellement. L'opérateur ' & ' indique le parallélisme

s_1 : $pause(vm_1)$; $pause(vm_2)$; $pause(vm_3)$;
($sauvegarde(vm_1)$ & $sauvegarde(vm_2)$ & $sauvegarde(vm_3)$ & $migration(vm_6)$)
s_2 : $migration(vm_7)$
s_3 : ($restaure(vm_4)$ & $restaure(vm_5)$) ;
(($relance(vm_4)$; $relance(vm_5)$) & $migration(vm_8)$)

FIGURE 9.2 – Plan de reconfiguration, modifié depuis la Figure 9.1, maintenant la cohérence des états dans les tâches j_1, j_2 et j_3. L'opérateur ' ; ' indique la séquentialité.

L'implémentation du changement de contexte est réalisée en modifiant l'implémentation du VMRP dé-

crit dans le chapitre 8. L'approche pour la résolution du VMRP à base de programmation par contraintes a facilité cette adaptation. En effet, il suffit simplement de supprimer la contrainte maintenant un nombre de nœuds minimum ; puis d'ajouter une contrainte dédiée à l'optimisation de la configuration résultat. Cette contrainte est semblable à la contrainte réduisant le coût de reconfiguration, mais l'algorithme pour la création de plans sous-jacent tient compte de la modification proposé ci-dessus.

9.2 Exemple d'implémentation d'un ordonnanceur

Nous avons appliqué notre architecture avec un exemple d'ordonnancement avant un objectif usuel : l'exécution des tâches au plus tôt. Ce type d'ordonnancement tente d'exécuter les différentes tâches soumises dès que possible pour que l'utilisateur récupère au plus tôt le résultat de l'exécution de ses tâches. Différentes techniques, reprenant entre autre les critères énoncés par Feitelson *et al.* [FRS+97] décrit dans le Chapitre 5, sont alors disponibles pour satisfaire cet objectif :

- utiliser la consolidation dynamique pour héberger simultanément plus de tâches. La migration des machines virtuelles en cours d'exécution permet de maintenir une configuration viable la plupart du temps. Cependant, en allouant des ressources selon les besoins des machines virtuelles, il n'est pas possible de prédire et d'éviter la saturation de la grappe. Il est donc nécessaire de suspendre des tâches de basse priorité pour résoudre des problèmes de surcharge de grappe ;
- un ordre de file non strict empêche qu'une tâche en attente de ressources pour son exécution bloque les tâches suivantes pouvant s'exécuter. En ce sens, une approche *First Fit* lance une tâche dès que ses ressources sont disponibles. Cependant, pour empêcher les risques de famine, la tâche bloquante doit tout de même pouvoir s'exécuter une fois la quantité requise de ressources disponible, quitte à suspendre les tâches de moindre priorité.

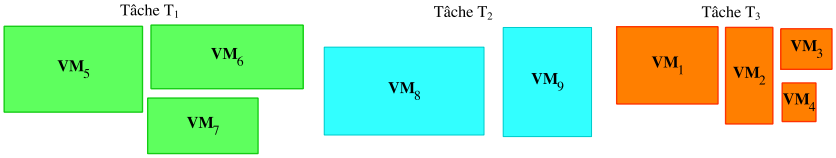
Le développement de cet algorithme d'ordonnancement entre dans la définition de la gestion dynamique de tâches à base de machines virtuelles. L'approche manipule en effet à la volée l'état des machines virtuelles ainsi que leur placement dans la grappe. Elle repose sur un modèle de tâches évolutives et un partitionnement à la fois spatial, pour la consolidation, et temporel, avec la préemption de tâches pour résoudre des problèmes d'accessibilité aux ressources.

Notre approche calcule la liste des tâches qui seront en état `Exécution` et la liste des tâches qui seront en état `Prêt` lors de la prochaine itération. Pour cela, l'algorithme utilise une file contenant la liste des tâches soumises triée et utilise un ordre non strict *First Fit*, en se servant de la date de soumission comme un indice de priorité. Le calcul de la liste des tâches à exécuter est alors réalisé par une heuristique dérivée de FFD :

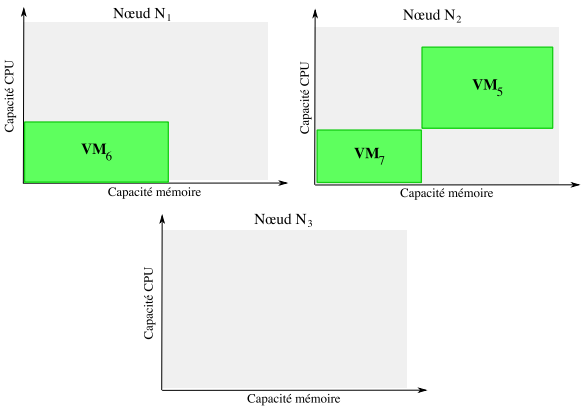
La file `Tâches` contient toutes les tâches considérées par l'ordonnanceur, classées par ordre de priorité. La liste `Exécutions`, initialement vide, contient la liste des tâches qui seront dans l'état `Exécution` lors de la prochaine itération. De même la liste `Prêts`, initialement vide, contient la liste des tâches qui seront dans l'état `Prêt` lors de la prochaine itération.

Pour chaque élément dans la file `Tâches`, nous vérifions avec l'heuristique FFD si une configuration viable composée des tâches de la liste `Exécutions` et de la tâche courante est possible. Si oui, alors la tâche est ajoutée à la liste `Exécutions`, sinon elle est rajoutée à la liste `Prêts`. Après le test du dernier élément de la file, la liste `Exécutions` contient un ensemble de tâche qui sont assurées de pouvoir s'exécuter sur la grappe.

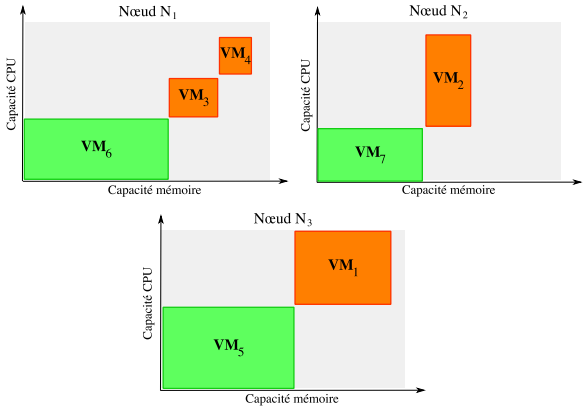
Un exemple de calcul des listes des tâches est décrit dans la Figure 9.3. La file `Tâches` contient les tâches T_1, T_2 et T_3. Une première itération permet de vérifier qu'il est possible d'exécuter la tâche T_1 (voir Figure 9.3(b)). Il n'est cependant pas possible d'exécuter simultanément les tâches T_1 et T_2 (la configuration résultat n'est pas viable). Finalement, il est possible de calculer une configuration viable exécutant simultanément les tâches T_1 et T_3 (voir Figure 9.3(c)). Si l'on suppose que les tâches T_1 et T_2 sont en cours d'exécution et que la tâche T_3 est en attente, alors le changement de contexte va suspendre T_2, lancer T_3 et migrer si nécessaire des machines virtuelles de la tâche T_1. Lorsque les besoins en ressources des machines virtuelles de T_1 seront suffisamment réduits pour exécuter la tâche T_2, celle-ci sera reprise quitte à suspendre T_3.



(a) Liste des tâches à parcourir. La tâche T_1 est la tâche la plus prioritaire.



(b) Une configuration viable impliquant la tâche T_1



(c) Une configuration viable impliquant les tâches T_1 et T_3

FIGURE 9.3 – Exemple de sélection des tâches à exécuter

9.3 Travaux apparentés

Différents travaux, tous récents, ont montré le manque de flexibilité des gestionnaires de ressources couramment utilisés et proposent des ordonnanceurs basés sur la manipulation de machines virtuelles.

Sotomayor *et al.* [SMLF08, SKF08] proposent avec *Haizea* le concept de location pour l'approvisionnement en ressources. Les utilisateurs négocient une quantité de ressources pour une durée finie et indiquent le mode de soumission de leurs tâches. Les tâches en mode *best-effort* sont moins prioritaires et peuvent être suspendues si nécessaire pour exécuter des tâches plus prioritaires où ayant réservées des ressources à l'avance. Cette approche permet également de renégocier le contrat de location des ressources, mais ne permet pas de s'adapter dynamiquement aux besoins réels courants des différentes tâches.

Grit *et al.* [GIYC06, GIMS07] proposent un gestionnaire de ressources permettant de développer des stratégies d'ordonnancement à base de règles et considèrent les actions de lancement et d'arrêt de tâches ainsi que l'action de migration des machines virtuelles en cours d'exécution. D'une façon similaire à Entropy, cette approche propose une séparation entre un module décisionnel pour l'ordonnancement des machines virtuelles et un module appliquant les transitions. Il n'est cependant pas possible d'implémenter des stratégies d'ordonnancement utilisant un partitionnement temporel puisque les actions de suspension et de reprise ne sont pas prises en compte, bien que nécessaires.

Fallenbeck *et al.* [FPSF06] proposent une approche non-invasive utilisant des gestionnaires de ressources standards. Cette solution réalise un partitionnement statique des nœuds de calcul en fonction de différentes files d'attente disponibles pour les utilisateurs. À chaque file d'attente de l'ordonnanceur est associée une partition dédiée à l'exécution de ses tâches. Sur chaque nœud de calcul s'exécute un hyperviseur pouvant exécuter plusieurs machines virtuelles pré-fabriquées. Chaque machine virtuelle correspond à un hôte potentiel pour une file d'attente. En fonction de la taille des différentes files d'attente, le gestionnaire de ressources sélectionne la machine virtuelle a lancer sur chaque hyperviseur. Ainsi, le système utilise toujours le partitionnement statique défini par l'ordonnanceur, mais fait varier le nombre d'hôtes de chaque partition afin de réduire le nombre de nœuds inutilisés lorsqu'une partition dédiée à une file d'attente est sous-exploitée. Notre solution diffère de par l'approche retenue pour nous adapter aux besoins : nous améliorons l'utilisation des ressources en agissant sur les tâches plutôt que sur leur

support d'exécution.

Plus généralement, les différentes approches décrites dans cette section mettent en avant un besoin de flexibilité dans la gestion des ressources et proposent une approche à base de machines virtuelles. Ces différentes approches se basent sur les mécanismes manipulant l'état et la position des machines virtuelles mais leurs possibilités sont plus limitées. De plus, ces différentes approches ne considèrent pas nécessairement la notion de changement de contexte comme un élément de base dégageant le développeur des préoccupations liées au changement de contexte.

9.4 Conclusion

Bilan

Nous avons discuté dans ce chapitre de l'utilisation d'Entropy comme un gestionnaires de ressources pour des tâches composées de machines virtuelles. L'administrateur développe sa propre stratégie d'ordonnancement en se concentrant uniquement sur le calcul d'une configuration viable indiquant l'état des différentes tâches pour l'instant suivant, tandis que le module de reconfiguration réalise un changement de contexte en modifiant l'état des tâches ainsi que la position des machines virtuelles en cours d'exécution afin d'assurer la transition entre la configuration courante et la nouvelle configuration. Cette approche permet aux administrateurs de développer des algorithmes complexes nécessitant de la gestion dynamique des tâches sans se préoccuper des problèmes liés à la reconfiguration. Notre approche à base de programmation par contraintes a facilitée l'adaptation du code du module de planification en composant un problème basé sur le VMAP en ajoutant ou en retirant des contraintes.

Nous avons également présenté dans ce chapitre un exemple d'implémentation d'un ordonnanceur exécutant des tâches au plus tôt en considérant un ordre de file non-strict basé sur *First Fit* et un partitionnement des ressources à la fois spatial et temporel pour des tâches évolutives. Appliquer les changements sur les tâches d'après un tel algorithme requiert d'utiliser les mécanismes de migration, de suspension et de reprise de machines virtuelles.

Perspectives

Le développement d'une stratégie d'ordonnancement est simple et rapide pour une personne connaissant l'API d'Entropy, mais elle nécessite un minimum de compétences dans le langage de programmation JAVA et dans l'interfacage avec Entropy. Ces deux points peuvent freiner les administrateurs dans le développement d'ordonnanceurs et mener à des erreurs de conception liées au manque d'expertise dans l'API d'Entropy et à l'absence de vérification à priori des algorithmes. En effet, certaines erreurs de conceptions ne seront observables que durant l'exécution du système. Une approche pour l'écriture d'ordonnanceur pour Entropy à base d'un langage dédié par exemple (DSL) pourrait être une solution à ces problèmes. Un DSL est un langage créée pour résoudre des problèmes spécifiques à un domaine. Le langage SQL [MS01] par exemple est un langage dédié à la manipulation de bases de données relationnelles. Ces langages s'opposent à des langages généralistes comme JAVA ou C qui couvrent un large spectre de domaines. Cette approche réduit le niveau d'expertise nécessaire à l'écriture de stratégies d'ordonnancement en supprimant les pré-requis liés au langage JAVA et en permettant la vérification de certaines propriétés de corrections avant son exécution. Le développeur manipule uniquement des concepts liés à l'ordonnancement des tâches et à la génération d'une configuration avec un langage dédié. Ce code est ensuite compilé puis incorporé dans Entropy uniquement s'il respecte des propriétés telles que l'impossibilité de détruire involontairement des tâches par exemple.

Le chapitre suivant de ce manuscrit est dédié aux évaluations des différentes contributions de cette thèse. Nous discutons des différentes expérimentations qui ont servi à valider le prototype Entropy et nos choix de conceptions tel que l'utilisation de la programmation par contraintes et l'optimisation du processus de reconfiguration.

Chapitre 10

Évaluation

Où nous évaluons les performances des modules de décision dédiés à la consolidation dynamique et à l'ordonnancement flexible de tâches, ainsi que les performances du module de reconfiguration. Les expérimentations réalisées au travers de simulations et de l'exécution de tâches sur des grappes démontrent l'applicabilité de notre approche, en compensant un temps de calcul des solutions supérieur aux approches à base d'heuristiques par une plus grande qualité des solutions et une forte réduction du temps de reconfiguration.

Sommaire

Nous avons supposé durant cette thèse que l'utilisation des machines virtuelles et d'un système autonome flexible était une solution viable pour une gestion dynamique des tâches. Nous pensons en effet que les mécanismes dédiés à la manipulation des machines virtuelles ainsi que la flexibilité dans l'écriture de module de décision et de planification permettent le développement de différentes politiques à base de gestion dynamique des tâches améliorant l'utilisation des ressources. Nous avons proposé dans le Chapitre 7 un premier cas d'utilisation dédié à la consolidation dynamique de machines virtuelles, puis nous avons proposé dans le Chapitre 9 une approche pour l'écriture de stratégies d'ordonnancement.

Dans cette thèse, notre approche se démarque des solutions actuelles à la fois par sa flexibilité, offerte par la programmation par contraintes, qui permet de définir des problèmes combinatoires complexes dédiés au calcul de configurations viables, mais également par la considération des problèmes liés à la manipulation de l'état et de la position des machines virtuelles dans les grappes. Le calcul de nouvelles configurations viables par un module de décision ainsi que la réduction du temps de reconfiguration par le module de planification sont cependant des problèmes complexes dont la résolution peut nécessiter un temps de calcul important. En contre-partie, les solutions calculées sont supposées être d'une qualité supérieure aux solutions ad-hoc actuelles.

Nous réalisons dans ce chapitre une évaluation d'Entropy afin de valider nos différentes hypothèses. Dans une première section, nous réalisons différentes micro-évaluations des modules de résolution des différents problèmes définis durant cette thèse, afin d'estimer leurs temps d'exécution et la qualité de leurs résultats. Ces premières évaluations servent également à calibrer Entropy en définissant un temps maximal pour la résolution de problèmes qui seront traités dans des évaluations sur des grappes. Dans une seconde section, nous évaluons le système Entropy dans son ensemble par différentes expérimentations sur des grappes, afin de vérifier l'efficacité de notre approche pour la consolidation dynamique et

l'ordonnancement de tâches. Nous concluons ce chapitre en discutant des limites de nos expérimentations et des solutions envisageables.

10.1 Micro-évaluations

Nous utilisons pour la résolution de nos différents problèmes une approche générique exacte. Cette méthode de résolution permet plus de flexibilité dans la définition des problèmes combinatoires, mais elle implique souvent un temps de résolution plus long comparé aux approches heuristiques habituelles. Il est donc nécessaire d'évaluer d'abord le temps moyen de résolution de notre approche pour les problèmes définis dans cette thèse pour ensuite comparer la qualité des résultats.

10.1.1 Protocole expérimental

Pour évaluer le temps de résolution des problèmes ainsi que la qualité des solutions, nous avons généré aléatoirement 100 configurations différentes mais partageant certaines propriétés basiques. Chaque nœud de calcul dispose de 3 Go de mémoire vive et une capacité CPU égale à 2. Chaque machine virtuelle a un besoin en ressource CPU égale à 0 ou 1.

Les configurations sont ensuite traitées comme des problèmes à résoudre dans un temps limite fixé arbitrairement à 60 secondes. Si passé ce délai, le solveur n'a pas prouvé que la dernière solution calculée est optimale, alors celui-ci retourne la meilleure solution calculée : celle-ci peut être une solution non-optimale, ou bien une solution optimale qui n'a pas encore été prouvée comme telle. Afin d'évaluer le temps de calcul moyen nécessaire pour résoudre un ensemble de configurations, nous nous intéressons à l'évolution du processus d'amélioration de la solution durant la résolution. Nous notons ainsi en fonction du temps le nombre de problèmes dont la meilleure solution à ce moment a été calculée (mais pas nécessairement prouvée). La machine utilisée pour cette évaluation dispose d'un processeur AMD Opteron cadencé à 2.0 GHz et de 4 Go de mémoire vive.

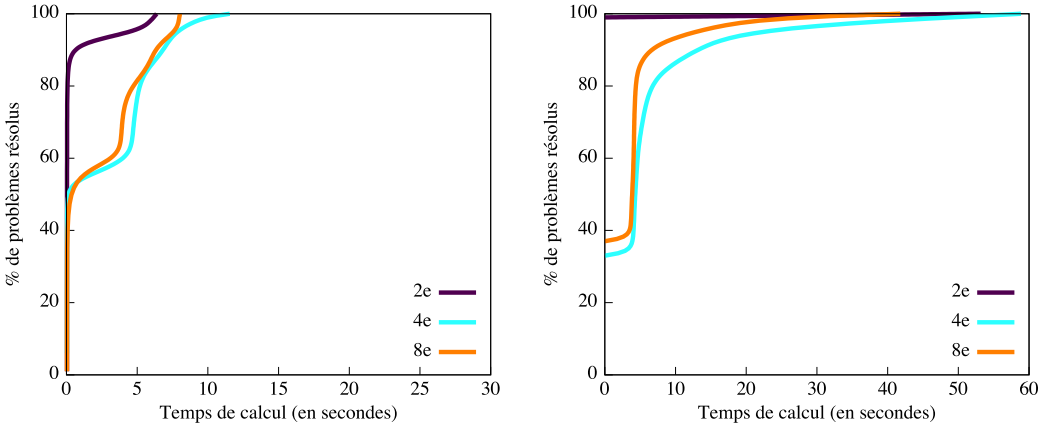
10.1.2 Critères impactant le temps de résolution des problèmes VMPP, VMRP

La complexité de la résolution des problèmes VMPP, VMRP et RJAP définis dans cette thèse respectivement dans les chapitres 7,8,9 dépend principalement du nombre de nœuds et de machines virtuelles impliqués, du nombre d'éléments équivalents (nœuds ou machines virtuelles) et du rapport entre le nombre de machines virtuelles et le nombre de nœuds. Pour évaluer notre approche de résolution, nous évaluons l'impact de ces spécificités sur le temps de résolution. Pour faciliter cette observation, nous nous concentrons sur les problèmes d'optimisation VMPP et VMRP. Nous ne considérons pas ici le problème de satisfaction RJAP qui, bien que la durée de sa résolution soit sensible aux même spécificités, est solvable plus rapidement. L'observation de l'impact des spécificités des configurations sur la résolution du problème est alors plus difficile.

Impact des symétries Afin de réduire le temps de résolution des problèmes VMPP, VMRP et RJAP nous utilisons une contrainte qui considère les nœuds et les machines virtuelles équivalentes pour réaliser un filtrage supplémentaire sur l'arbre de recherche. Dans une configuration, une classe d'équivalence correspond à un type possible de machines virtuelles ou de nœuds. Le nombre de classes d'équivalence pour les machines virtuelles d'une configuration correspond au nombre de valeurs différentes pour les besoins CPU de toute les machines virtuelles multiplié par le nombre de valeurs différentes pour les besoins mémoire. Ainsi, une configuration ayant 2 classes d'équivalence pour les machines virtuelles peut disposer par exemple de machines virtuelles ayant un besoin en ressources mémoire égale à 1 Go et un besoin en ressource CPU égal à 0 ou 1.

Pour observer l'impact des symétries sur le temps de résolution lié à une présence plus ou moins forte de symétries, nous générons 3 ensembles de configurations composées de 200 machines virtuelles et de 200 nœuds mais dont les nombres de classes d'équivalence pour les machines virtuelles diffèrent : l'ensemble `2e` est composé de configurations ayant 2 classes d'équivalence pour les machines virtuelles (chacune d'entre elles requiert 1 Go de mémoire vive) ; l'ensemble `4e` utilise des configurations ayant 4 classes d'équivalence pour les machines virtuelles (chacune d'entre elles requiert 1 Go ou 2 Go de mémoire vive), l'ensemble `8e`

est composé de configurations ayant 8 classes d'équivalence pour les machines virtuelles (chacune d'entre elles requiert 512 Mo, 1 Go, 1.5 Go ou 2 Go de mémoire vive).



(a) VMPP (b) VMRP

FIGURE 10.1 – Impact des classes d'équivalences des machines virtuelles sur le temps de résolution de problèmes VMPP et VMRP.

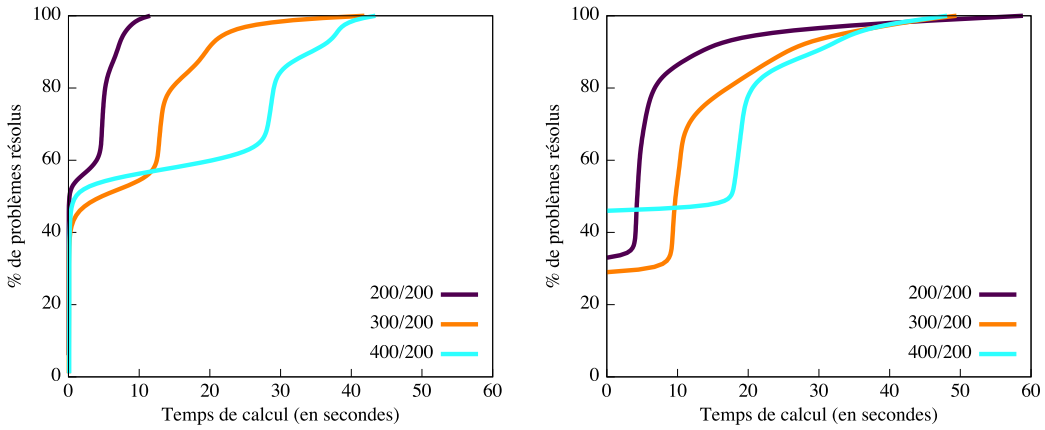
La Figure 10.1(a) décrit le pourcentage de configurations de chaque ensemble résolues avec le VMPP en fonction du temps. Conformément à notre intuition, le nombre de classes d'équivalence affecte le temps nécessaire à la résolution d'un problème : alors que 6 secondes sont nécessaires pour calculer la valeur minimum pour 100% des configurations de l'ensemble `2e`, 7 secondes sont nécessaires avec l'ensemble `4e` et 11 secondes avec l'ensemble `8e`.

La Figure 10.1(b) représente le pourcentage de configurations de chaque ensemble résolues avec le VMRP. Ce problème est de par sa nature plus complexe à résoudre que VMPP et le nombre de solutions est considérablement supérieur aux solutions de VMPP. Le solveur ne parvient pas parfois à prouver que la solution est optimale durant les 60 secondes allouées. De plus, le gain relatif est moins significatif pour VMRP : le coût d'un plan de reconfiguration peut être de l'ordre de plusieurs centaines de milliers et, contrairement à VMPP, améliorer ce coût d'une dizaine d'unités n'est pas intéressant dans la pratique. Nous considérons alors qu'une solution est minimale jusqu'à ce que le solveur ne calcule une nouvelle solution avec un coût plus petit d'au moins 10%. Le graphe 10.1(b) montre que le processus de résolution de VMRP est également impacté par le nombre de classes d'équivalence : 8 secondes sont nécessaires pour résoudre 90% des configurations de l'ensemble `4e` tandis que 15 secondes sont nécessaires avec l'ensemble `8e`.

D'après ces résultats, nous concluons que le temps de résolution de VMPP et de VMRP doit être augmenté lorsque le nombre de classes d'équivalence augmente. En effet, pour un nombre d'éléments donné, une configuration avec un nombre élevé de classes d'équivalence possède peu d'éléments dans chaque classe. La contrainte manipulant les symétries est alors moins activée et diminue moins la taille de l'arbre de recherche. Dans cette situation, le temps nécessaire au calcul d'une « bonne » solution augmente.

Impact du ratio de machines virtuelles par nœud Nous avons généré 3 ensemble de configurations, toutes ayant 4 classes d'équivalence pour les machines virtuelles. Chaque ensemble est constitué de configurations composées de 200 nœuds mais dont le nombre de machines virtuelles varie : l'ensemble `200/200` utilise 200 machines virtuelles, l'ensemble `300/200` utilise 300 machines virtuelles et l'ensemble `400/200` utilise 400 machines virtuelles.

Les Figures 10.2(a) et 10.2(b) décrivent respectivement le pourcentage de problèmes VMPP et VMRP résolus pour chaque ensemble en fonction du temps. Nous observons que le nombre de machines virtuelles rapporté au nombre de nœuds impact sur le processus de résolution. Pour le VMPP, 10 secondes sont nécessaires pour résoudre 90% de l'ensemble `200/200`, 20 secondes sont nécessaires pour résoudre 90% de l'ensemble `300/200` tandis que 35 secondes sont nécessaires pour résoudre 90% des configurations de



(a) VMPP (b) VMRP

FIGURE 10.2 – Impact du ratio entre le nombre de machines virtuelles et le nombre de nœuds sur le temps de résolution des problèmes VMPP et VMRP.

l'ensemble `400/200`. Pour le VMRP, 90% des configurations de l'ensemble `200/200` sont résolues en 13 secondes, 25 secondes pour l'ensemble `300/200` et 25 secondes pour l'ensemble `400/200`.

L'impact du ratio de machines virtuelles par nœud s'explique par la difficulté de trouver un nœud disposant de suffisamment de ressources CPU et mémoire libres pour héberger chaque machine virtuelle. Lorsque la proportion de machines virtuelles par rapport aux nœuds augmente, il est naturellement plus difficile de calculer une configuration viable puisque l'ensemble des nœuds pouvant accepter une machine virtuelle est réduit (il y a moins de places libre dans la grappe). Concernant la résolution de VMRP, un ratio important de machines virtuelles par nœud implique une planification des actions de plus en plus complexe : le nombre de dépendances séquentielles augmente ainsi que la fréquence des cycles de dépendances. Il se peut également qu'il n'existe pas de plan de reconfiguration entre la configuration initiale et la configuration voulue. Lorsque le ratio de machines virtuelles par nœud augmente, il est donc nécessaire d'augmenter le délai maximal pour la résolution du VMRP afin de calculer de « bonnes » solutions.

Nous avons pu déterminer grâce à ces deux premières évaluations le temps de calcul nécessaire au calcul de « bonnes » solutions pour les problèmes VMPP et VMRP avec Entropy. Ce temps est variable et dépend de différents critères tels que le nombre de classes d'équivalence et le ratio de machines virtuelles par nœud. Comparé à une heuristique telle que FFD, notre approche à des temps de résolution significativement plus long. Les temps de résolution de l'heuristique FFD sont également soumis aux mêmes critères, cependant ils restent en dessous d'une seconde. Mis à part les considérations de flexibilité dans le développement de nos problèmes, notre approche ne peut être justifiée que si la qualité de nos solutions est supérieure à celle obtenue avec une heuristique ad-hoc.

10.1.3 Comparaison qualitative avec l'heuristique FFD

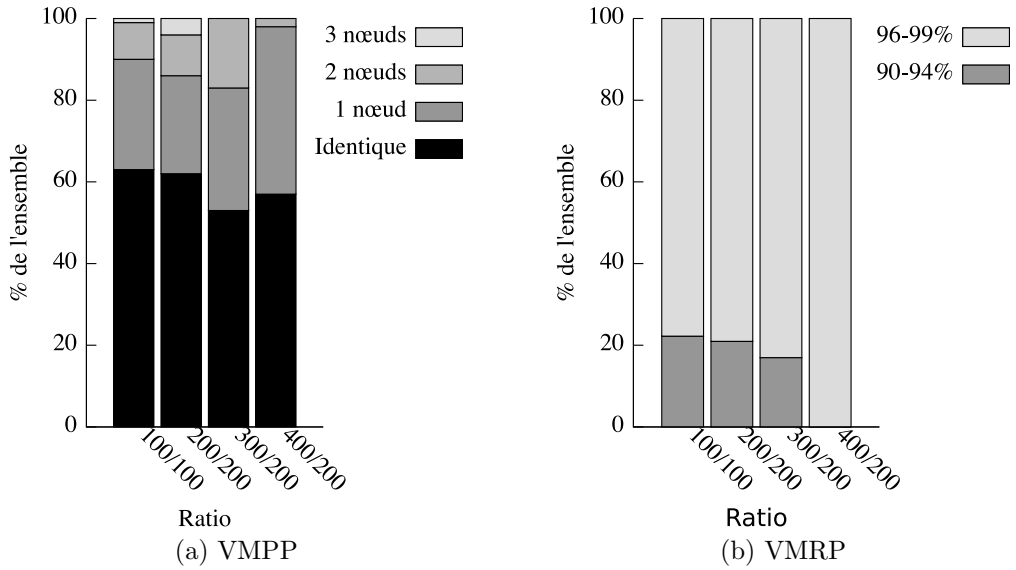
Pour cette évaluation, nous générons 4 ensembles de configurations, chacun avec 4 classes d'équivalence pour les machines virtuelles, en faisant varier le nombre de nœuds et de machines virtuelles. Nous comparons au travers de la résolution de ces configurations, la qualité des résultats de notre approche avec ceux obtenus par l'heuristique FFD, couramment utilisée dans des solutions de consolidation dynamique de machines virtuelles [BKB07, VAN08b, WSVY07]. Pour le VMPP, nous comparons le nombre de nœuds utilisés pour héberger des machines virtuelles avec la configuration calculée par FFD. Pour le VMRP, nous comparons le coût des plans associés aux configurations calculées. Pour chaque ensemble de configurations, nous limitons le temps maximal de résolution d'Entropy en considérant les temps observés durant les précédentes évaluations (voir Table 10.1).

La Figure 10.3(a) illustre l'amélioration des solutions d'Entropy comparées aux solutions calculées

Ensemble	VMPP	VMRP
100/100	5	10
200/200	10	13
300/200	20	25
400/200	35	30

TABLE 10.1 – Délais (en secondes) accordés à Choco pour résoudre les problèmes VMPP et VMRP en fonction des ensembles de configurations

avec l'heuristique FFD pour le VMPP. Nous observons d'abord que nos solutions n'utilisent jamais un nombre de nœuds supérieur aux solutions calculées par FFD. Ceci s'explique par la déclaration de la borne supérieure de la variable X, comptabilisant le nombre de nœuds, qui est égale à la solution calculée par FFD. Pour 42% des configurations, Entropy calcule des configurations strictement meilleures nécessitant jusqu'à 3 nœuds de moins qu'une solution calculée avec l'heuristique FFD. L'intérêt de cette amélioration est cependant à relativiser à la vue du nombre de nœuds des configurations. La Figure 10.3(b) décrit l'amélioration que réalise Entropy concernant le problème VMRP. Pour ce problème, nous devons tenir compte du fait que certaines solutions du VMPP avec Entropy ont un nombre de nœuds inférieur à FFD. Pour réaliser une comparaison équitable, nous ne considérons donc que les configurations pour lesquelles les solutions de VMPP sont équivalentes entre Entropy et FFD. Nous observons d'après ce graphe que Entropy calcule une nouvelle configuration dont le coût de reconfiguration associé est toujours réduit d'au moins 90%. Cette réduction s'améliore en même temps que le ratio de machines virtuelles par nœud augmente. Ainsi, pour l'ensemble `400/200`, 100% des solutions ont un coût inférieur à celui de FFD d'au moins 96%. Cette différence provient de l'approche de résolution exacte qui cherche une meilleure solution jusqu'à prouver que celle-ci est optimale ou que le délai maximal pour la résolution soit écoulé. À l'opposé, l'heuristique FFD s'arrête à la première solution, sans considérer la notion d'amélioration du coût de reconfiguration.



(a) VMPP (b) VMRP

FIGURE 10.3 – Comparaison qualitative des solutions entre Entropy et FFD

Si l'intérêt d'utiliser Entropy pour réduire le coût d'un plan de reconfiguration est évident par rapport à l'utilisation de la configuration calculée par FFD, l'intérêt d'utiliser Entropy pour résoudre VMPP paraît plus discutable. Il faut cependant considérer que notre approche de résolution ne calcule pas uniquement une solution utilisant le moins de nœuds possible, mais elle réduit également le nombre de migrations, grâce à nos heuristiques de choix de valeur. De précédentes expérimentations ont montré que

le remplacement de VMPP par l'heuristique FFD ne permettaient pas de réduire aussi efficacement le coût de reconfiguration du plan calculé par VMRP. En effet, le coût du plan associé à la configuration retournée par VMPP est utilisé comme borne supérieure du VMRP. Moins cette borne est proche de la valeur optimale, plus l'espace de recherche est grand et plus il est difficile d'optimiser le problème de la reconfiguration.

Cette évaluation met en évidence la viabilité de notre approche pour le calcul de plans de reconfiguration de faible coût tout en obtenant une configuration finale avec une qualité égale ou supérieure à celle calculée par l'heuristique FFD. Cependant, encore une fois, le temps de calcul nécessaire pour obtenir une configuration viable avec un nombre de nœuds réduit et dont le plan de reconfiguration à un coût réduit est très long comparé à l'heuristique FFD. En réalisant des expérimentations sur des grappes de calculs, nous pouvons vérifier si ce temps de calcul plus important est compensé par une réelle réduction du temps d'exécution du plan de reconfiguration.

10.2 Expérimentations sur une grappe

Nous réalisons dans cette section différentes évaluations de nos éléments de contribution sur des grappes de calculs. En exécutant différentes applications de la suite logicielle NASGrid [FdW02], nous évaluons les performances d'Entropy lors de l'exécution du module de consolidation dynamique et du module dédié à l'ordonnancement. Nous vérifions ainsi l'intérêt de la gestion dynamique des tâches réalisée par Entropy et son impact sur l'utilisation des ressources de la grappe.

10.2.1 La suite de tests NASGrid

L'évaluation sur grappe des différents modules d'Entropy consiste à analyser l'exécution de différentes tâches dont les besoins en CPU varient au cours du temps. L'implémentation actuelle d'Entropy se base sur des hyperviseurs ne proposant pas le partage de pages mémoires, nous considérons donc les besoins mémoire constants.

Nous avons choisi d'exécuter dans nos tâches la suite d'applications NASGrid 3.0 [FdW02], une suite logicielle développée par la NASA pour estimer les performances d'une grappe de serveurs par l'exécution d'applications scientifiques distribuées. Une application est définie par un graphe de précédence acyclique. Chaque nœud d'un graphe est un processus JAVA exécutant un calcul scientifique (une transformée de Fourier par exemple). Nous embarquons chacun de ces processus dans une machine virtuelle exécutant en mode paravirtualisation une distribution GNU/Linux. Les arcs du graphe indiquent le séquencement des calculs. Ainsi, un calcul est lancé lorsque tous les calculs précédents ont été terminé. Au démarrage de l'application, plusieurs calculs peuvent être lancés simultanément, une fois les derniers calculs exécutés un rapport indique le temps de calcul global de l'application. L'exécution d'une application passe donc par différentes phases, définies par les calculs en cour d'exécution. Une tâche exécutant une telle application peut donc être considérée comme une tâche évolutive. Lorsqu'un processus exécute un calcul scientifique, celui-ci nécessite toute la capacité disponible d'un CPU, nous considérons alors que son besoin CPU est égal à 1. Dans le cas contraire, les besoins CPU sont négligeables.

En standard, la suite NASGrid propose 4 applications. Ces applications représentent des graphes communs dans les applications de simulation de phénomènes physiques. Pour chaque application, il est possible de choisir la taille des calculs à effectuer. Par exemple, les calculs de la taille S nécessitent un temps d'exécution et une quantité de mémoire inférieurs aux calculs de la taille W. La Figure 10.4 décrit les graphes des 4 applications de base. L'application ED (*Embarrasely Distributed*) représente une classe commune dans les applications grilles, où un même calcul est réalisé plusieurs fois, en parallèle, mais avec des données en entrée différentes. L'application HC (*Hielical Chain*) représente une chaîne, où chacun est exécuté à la suite du précédent. VP (*Visualisation Pipe*) représente une chaîne de processus composites tels que celle rencontrée lorsque l'on visualise la progression de la résolution d'un flot de calculs. Finalement, MB (*Mixed Bag*) représente un type d'application similaire à VP mais introduit une asymétrie dans les différents calculs : les durées de calculs et les quantités de données transférées entre chaque calcul varient.

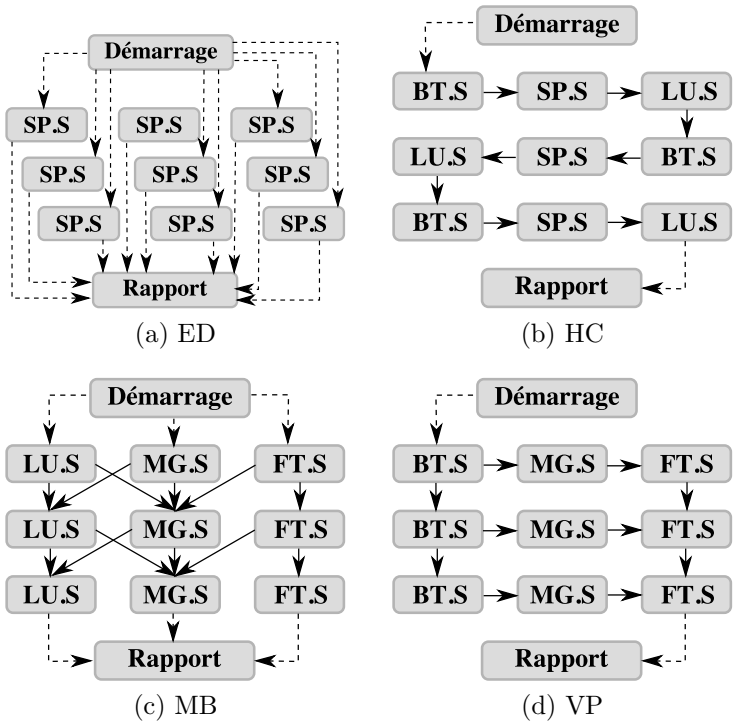


FIGURE 10.4 – Graphes de calculs standard de la suite NASGrid

10.2.2 Consolidation dynamique

Cette expérience porte sur l'utilisation d'Entropy pour réaliser de la consolidation dynamique. Cette évaluation a été réalisée dans le cadre de l'expérience Grid'5000 [BCC+06], développée autour de l'action INRIA ALADDIN et avec, entre autres, le support du CNRS et de RENATER. Grid'5000 est une grille informatique regroupant différentes grappes hébergées sur 11 sites géographiques en France.

Notre grappe de test est constituée de 39 nœuds interconnectés par un réseau Ethernet Gigabit. Chacun dispose d'un processeur AMD Opteron cadencé à 2 GHz et de 2 Go de mémoire vive. 35 nœuds sont des nœuds de calculs exécutant un hyperviseur Xen 3.0 et réservant 200 Mo de mémoire pour le Domaine-0. Leur capacité CPU est égal à 1. 3 nœuds sont dédiés au stockage des images des disques des machines virtuelles. Finalement, un nœud de service exécute Entropy. Les différents nœuds de calculs hébergent 35 machines virtuelles exécutant des applications semblable à ED, HC et VP, chaque application dispose de son propre serveur de fichiers. L'application ED utilise 10 machines virtuelles avec 512 Mo de mémoire chacune. L'application HC utilise 5 machines virtuelles avec 764 Mo de mémoire chacune et l'application VP utilise 20 machines virtuelles avec 512 Mo de mémoire chacune. L'application MB n'est pas utilisée pour cette évaluation : son exécution requiert une quantité de mémoire vive trop importante, limitant les possibilités de consolidation.

Avant de démarrer l'expérience, toutes les machines virtuelles sont lancées et prêtes à exécuter les applications, leur besoin en ressources CPU est alors égale à 0. La configuration initiale correspond alors à une consolidation maximale sur 13 nœuds calculée par Entropy. Les 3 applications sont ensuite lancées simultanément. Dans une première expérience, nous exécutons Entropy avec le module de décision dédié à la consolidation dynamique. Nous réalisons ensuite le même test en utilisant une consolidation dynamique basée sur l'heuristique FFD.

Dans la Figure 10.5 chaque point représente le coût calculé et la durée mesurée de chaque plan produit par Entropy ou avec FFD. La relation entre le coût d'un plan et sa durée d'exécution est grossièrement linéaire, la fonction de coût K apparaît donc comme un indicateur de durée d'une reconfiguration viable. Nous observons que les plans calculés par Entropy ont un coût et une durée d'exécution bien inférieurs aux plans calculés avec l'heuristique FFD. Le temps moyen d'exécution des plans pour FFD est de 6 minutes

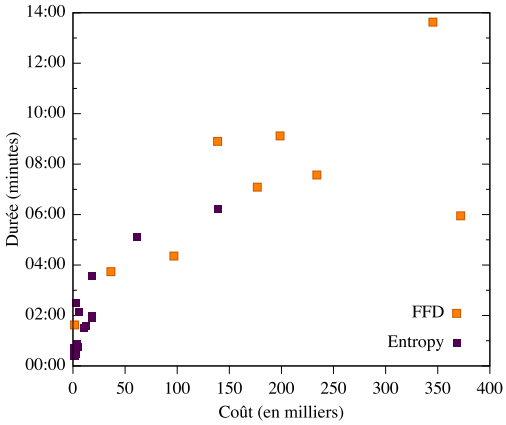
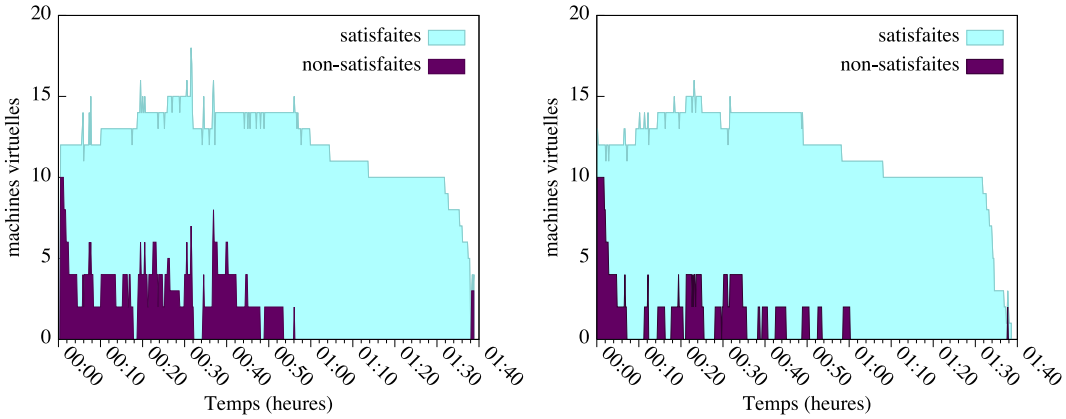


FIGURE 10.5 – Comparaison de la qualité de la solution de VMRP rapportée à FFD

53 secondes tandis que ce temps est ramené à seulement 1 minute et 47 secondes avec Entropy. Avec des plans de reconfiguration s'exécutant rapidement, Entropy réalise plus d'auto-adaptations : durant l'expérience, Entropy a réalisé 18 reconfigurations contre 9 lors de l'utilisation de FFD. Ce nombre de reconfigurations est important car, comme nous l'avons décrit initialement dans l'architecture de Entropy, une reconfiguration a lieu uniquement lorsque le module de décision détecte une configuration courante non-viable ou calcul une nouvelle configuration viable considérée comme meilleure. Les reconfigurations effectuées sont donc toujours nécessaires.



(a) FFD (b) Entropy

FIGURE 10.6 – Activité des machines virtuelles

Les Figures 10.6(a) et 10.6(b) décrivent l'activité des machines virtuelles durant les expériences. Nous considérons qu'une machine virtuelle est satisfaite lorsque ses besoins en ressources CPU sont différents de 0 et que son placement satisfait ses besoins. Si toute les machines virtuelles sont satisfaites, alors la configuration courante est viable. Si l'une des machines virtuelles n'est pas satisfaite, alors la configuration courante n'est pas viable. Nous observons que le nombre de machines virtuelles insatisfaites est inférieur lorsque l'expérience est réalisée avec Entropy : le nombre moyen de machines virtuelles insatisfaites est égal à 1.75 avec FFD contre 1.05 avec Entropy. Ce critère qualificatif décrit l'efficacité d'un système d'auto-optimisation puisque une configuration non-viable implique nécessairement une perte de performances.

Lorsque l'expérience démarre, 12 machines virtuelles exécutent un calcul au même moment. Entropy réarrange rapidement celles-ci et obtient une configuration viable à la minute 7. Avec l'heuristique FFD, la configuration courante ne sera pas redevenu viable avant la minute 18. À la minute 10, le nombre de machines virtuelles avec un besoin CPU égal à 1 augmente et crée une configuration non-viable. À ce moment, Entropy n'est pas en cours de reconfiguration, il calcule ainsi une nouvelle configuration,

ré-optimise l'agencement des machines virtuelles et obtient une configuration viable à la minute 11. FFD par contre est toujours en cours de reconfiguration à la minute 10, les besoins des machines virtuelles ayant changé, les mouvements qu'il réalise pour obtenir une configuration viable ne sont plus d'actualité, la configuration obtenue n'est alors plus viable jusqu'à la minute 18. Dans cette situation, nous considérons qu'une auto-optimisation avec l'heuristique FFD nécessite trop de temps comparée aux variations de l'activité des machines virtuelles. La Figure 10.6(b) montre qu'il n'y a plus de machines virtuelles insatisfaites après 1 heure. Ceci s'explique par la durée inégale des différentes applications. À la minute 50, l'application HC termine son exécution. L'activité de VP change aux minutes 54 et 58 et requiert une reconfiguration. Ensuite, il n'y a plus de variations dans l'activité des machines virtuelles pouvant entraîner une configuration non-viable. À 1 heure 10 minutes, l'application VP se termine, seule l'application ED est encore en fonctionnement et son activité est constante.

Le temps de réponse d'un processus de reconfiguration mesure la durée entre la détection d'une configuration non viable et la prochaine configuration viable. Il indique la capacité d'un processus de reconfiguration à s'adapter à l'activité des machines virtuelles. Pour cette expérience, le temps de réponse moyen pour FFD est de 248 secondes. Avec Entropy, le temps de réponse moyen est de 142 secondes.

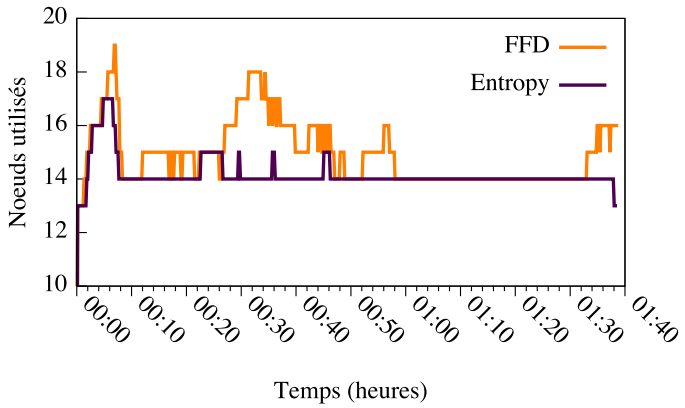


FIGURE 10.7 – Nombre moyen de nœuds utilisés avec FFD et Entropy

La Figure 10.7 décrit le nombre de nœuds utilisés pour héberger des machines virtuelles. Les plans de reconfiguration calculés avec FFD demandent plus de migrations. Ils contiennent potentiellement plus de cycles voire même des migrations inutiles comme la migration de toute les machines virtuelles d'un nœud vers un autre nœud équivalent. Durant le processus de reconfiguration, un nombre important de nœuds peut donc héberger temporairement des machines virtuelles. Pour cette expérience FFD a utilisé jusqu'à 4 nœuds supplémentaires durant une reconfiguration comparé à Entropy. Cette situation limite l'intérêt de la consolidation dynamique avec FFD dans une optique d'économie d'énergie. Si l'on éteint systématiquement tous les nœuds inutilisés, alors il peut être nécessaire d'allumer des nœuds supplémentaires pour réaliser un pivot ou des migrations inutiles. Entropy crée des plans de reconfiguration plus petits et limite, par la réduction du nombre total de migrations, le nombre de migrations inutiles. Entropy utilise alors un nombre de nœuds plus stable, son utilisation dans un contexte d'extinction des nœuds semble alors justifiée.

La consolidation dynamique implique une baisse de performance due au temps de reconfiguration et à la possibilité d'introduire des machines virtuelles insatisfaites lorsque le système n'est pas assez réactif. En minimisant le temps de reconfiguration, nous réduisons le temps où la configuration courante n'est pas viable, et donc la perte de performance associée à un mauvais agencement des machines virtuelles. La Figure 10.8 décrit le temps d'exécution de chaque application avec FFD, avec Entropy et avec un environnement sans consolidation. Dans ce dernier cas, chaque machine virtuelle est définitivement hébergée sur son propre nœud afin d'éviter toute perte de performance liée au partage des ressources CPU. Dans cette configuration, 35 nœuds sont donc nécessaires. Grâce à cet environnement de référence, nous calculons le surcoût en temps d'exécution lié à la consolidation dynamique. Pour cette expérience, ce surcoût est de 19.2% avec FFD. Entropy réduit ce surcoût à 11.5%.

Finalement, nous pouvons résumer le taux d'occupation des ressources pour les différentes applica-

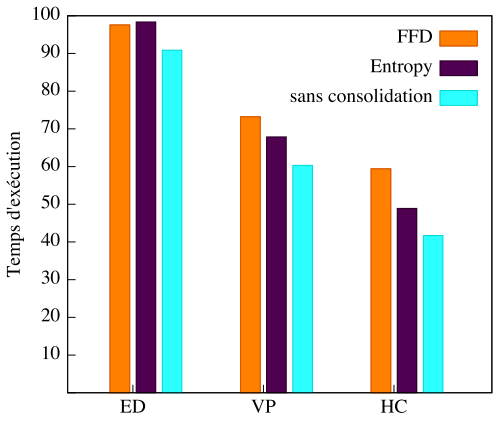


FIGURE 10.8 – Temps total d'exécution des applications.

tions par la quantité de nœuds nécessaire pour exécuter les applications en une heure. Dans le cadre de la consolidation dynamique, une quantité réduite implique une utilisation efficace des ressources. Sans consolidation, exécuter les différentes applications requiert 53.01 nœuds/heure. La consolidation dynamique avec FFD réduit cette consommation à 24.53 nœuds/heure. La consolidation dynamique réalisée par Entropy réduit cette consommation à 23.21 nœuds/heure. Cette mesure est cependant affectée par la durée de chaque application. Lorsque toute les applications sont en cours d'exécution, la consolidation provient uniquement du mélange des différentes machines virtuelles des applications. Lorsqu'une application se termine, cela crée des machines virtuelles « zombies » , qui consomment inutilement de la mémoire sur les nœuds. Ainsi pour ne considérer que la consommation des ressources liée à la consolidation dynamique, nous estimons celle-ci uniquement lorsque toutes les applications sont en fonctionnement, c'est à dire jusqu'à la fin de l'application HC. Dans cette situation, exécuter les 3 applications consomme 24.31 nœuds/heure sans consolidation. Avec FFD, cette consommation est réduite à 15.34 nœuds/heure. Entropy réduit cette consommation à 11.72 nœuds/heure.

Pour résumer, cette expérience a montré la viabilité de notre module de décision dédié à la consolidation dynamique. Comparé à l'utilisation de l'heuristique FFD, nous réduisons la durée moyenne d'exécution d'un plan de reconfiguration de 74%. Cette optimisation du processus de reconfiguration réduit le nombre de nœuds/heure de 50% comparée à une exécution native pour un surcoût à l'exécution de 11.5%.

10.2.3 Ordonnancement flexible de tâches

Cette expérience a pour objectif de valider la faisabilité de notre module de décision dédié à l'ordonnancement de tâches en exécutant sur une grappe une suite de tâches, prises en charge par la stratégie d'ordonnancement décrite dans le Chapitre 9.

La grappe utilisée pour cette évaluation est composée de 15 nœuds, 11 sont des nœuds de calculs, exécutant un hyperviseur Xen 3.2. Chacun de ces nœuds est équipé d'un processeur Intel Core 2 Duo (un CPU contient deux cœurs) cadencé à 2.1 GHz et de 3.6 Go de mémoire vive utilisable pour les machines virtuelles. Leur capacité CPU est fixé à 2. Trois nœuds exportent les images des disques des machines virtuelles par le biais de serveurs NFS et un nœud de service avec un processeur Intel Core 2 Duo cadencé à 2 GHz et disposant de 4 Go de mémoire vive exécute Entropy (un seul cœur est utilisé). Tous ces nœuds sont interconnectés par un réseau Ethernet Gigabit.

L'expérience consiste à exécuter 8 tâches, chacune composée de 9 machines virtuelles. Chaque tâche est soumise au même moment mais dans un ordre définie. Chaque tâche exécute une application NASGrid qui est démarrée une fois que toutes les machines virtuelles de la tâche sont lancées. Lorsque l'application est terminée, celle-ci demande à l'ordonnanceur de détruire sa tâche. Chaque machine virtuelle utilise une quantité de mémoire constante, comprise entre 512 Mo et 2048 Mo et requiert une capacité CPU de 1 lorsque le composant de l'application NASGrid exécute un calcul, et 0 sinon.

La Figure 10.9 décrit le coût des différents changements de contexte réalisés durant l'expérience. Des changements de contexte avec un coût réduit exécutent uniquement des actions de migrations, de lancement et d'arrêt. Par exemple, les 5 changements de contexte de coût égal à 0 exécutent uniquement des actions de lancement et d'arrêt et ont été exécutées en 13 secondes au plus. Le changement de contexte avec un coût de 1024 exécute 3 migrations en 19 secondes. Les changements de contexte avec un coût plus élevé exécutent en plus des actions décrit précédemment, des actions de reprise et de suspension. Par exemple, le changement de contexte avec un coût de 4068 a nécessité 5 minutes et 15 secondes pour exécuter 9 actions d'arrêt, 18 actions de lancement, 9 actions de reprise et 9 migrations. Contrairement à l'expérience précédente, la durée d'exécution d'un plan n'est pas nécessairement proportionnelle à son coût. Nous avons en effet décrit dans le chapitre 8 que les actions de migration, de suspension et de reprise peuvent avoir un même coût, même si le temps d'exécution d'une reprise peut être jusqu'à 6 fois plus long que le temps d'une migration d'une machine virtuelle identique. Nous observons cependant que notre fonction de coût est une solution viable pour favoriser le localisme : 21 des 28 actions de reprises exécutées durant cette expérience ont été des actions de reprise locale.

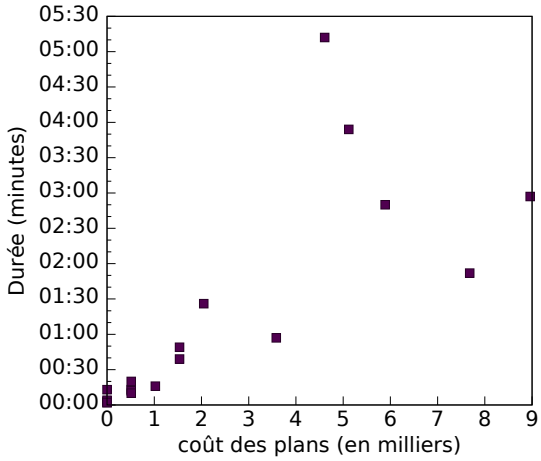


FIGURE 10.9 – Durée d'exécution des changements de contexte en fonction de leur coût

La seconde partie de cette expérience montre le bénéfice à utiliser notre module de décision dédié à l'ordonnancement, basé sur une gestion dynamique des tâches, comparé à une allocation statique des ressources. Pour cela, nous simulons l'exécution des différentes tâches avec un algorithme de type FCFS utilisé couramment dans différentes grappes en production [ET05]. La Figure 10.10 décrit son diagramme d'exécution en précisant le temps d'exécution de chaque tâche. Avec un tel algorithme, la partition des ressources pour chaque tâche est variable ; l'agencement de celles-ci est optimal et réserve en permanence 18 CPU sur les 22 disponibles. Comme toute les tâches ont les même besoins CPU et que leur exécution n'est pas limitée par la mémoire disponible sur les nœuds, alors une tâche ne peut pas bloquer inutilement d'autres tâches et un algorithme de *backfilling* n'est pas nécessaire.

Les Figure 10.11(b) et 10.11(a) décrivent le taux d'utilisation des ressources mémoire et CPU en utilisant l'algorithme FCFS ou notre algorithme. Nous observons que le taux d'utilisation des ressources avec notre algorithme est le plus important jusqu'à la minute 30. Ensuite les ressources sont globalement de moins en moins utilisées car il n'y a plus de tâches en attente d'exécution. Dans le cadre d'une exécution avec Entropy, 2 minutes 10 secondes après le début de l'expérience, la grappe est surchargée : les tâches en cours d'exécution demandent une capacité CPU de 29 alors que la grappe dispose d'une capacité CPU totale de 22. Dans cette situation, le module de décision calcule une nouvelle configuration et indique les nouvelles tâches qui doivent être exécutée afin d'obtenir une configuration viable. Le changement de contexte suspend alors les tâches n'appartenant plus à cette solution.

Le module de décision dédié à l'ordonnancement a permis de développer simplement un algorithme d'ordonnancement réalisant une gestion dynamique des tâches. Par comparaison avec la stratégie d'ordonnancement usuelle FCFS, notre module améliore le taux d'utilisation des ressources en lançant les différentes tâches au plus tôt, tout en résolvant les situations où la grappe est surchargée. Cette meilleure

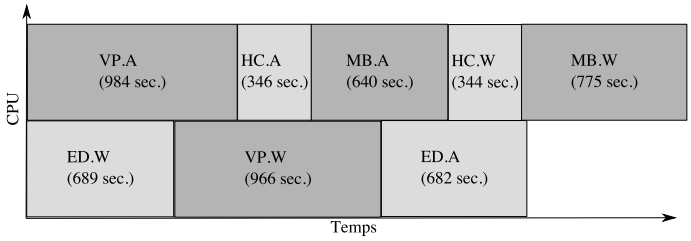
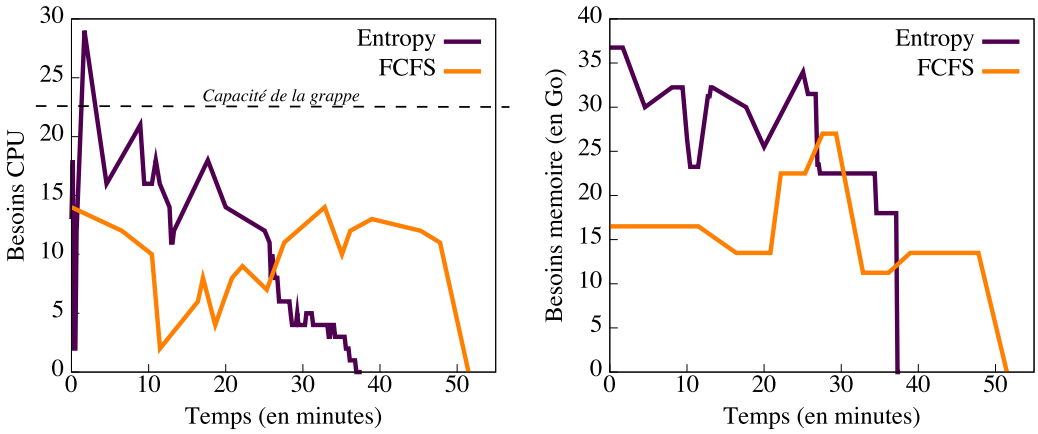


FIGURE 10.10 – Diagramme d'exécution des tâches avec un algorithme FCFS



(a) Ressources CPU (b) Ressources mémoire

FIGURE 10.11 – Utilisation des ressources

gestion des ressources permet d'exécuter la liste de tâches plus rapidement. En effet, le temps total nécessaire pour exécuter toutes les tâches, en comptant leurs temps d'attente est de 250 minutes avec l'algorithme FCFS. Avec notre stratégie d'ordonnancement, le temps total est de 150 minutes, soit une réduction de 40%.

10.3 Conclusion

Bilan

Nous avons montré dans ce chapitre aux travers de diverses expérimentations que les choix proposés dans cette thèse pour la gestion dynamique des tâches et l'amélioration de l'utilisation des ressources dans les grappes étaient pertinents. Notre approche permet le développement de stratégies utilisant une gestion dynamique des tâches qui améliorent l'utilisation des ressources des grappes comparé à des approches standards tout en impliquant un impact sur les performances raisonnable. Nous avons évalué les différents modules composant notre prototype Entropy en mesurant d'abord l'impact de différents critères sur le processus de résolution des problèmes VMPP et VMRP ; ainsi que la qualité des configurations calculées en terme respectivement de nombre de nœuds et de temps de reconfiguration. La comparaison de ces résultats avec ceux obtenus par l'heuristique FFD a montré que notre approche améliore sensiblement la qualité des configurations obtenues avec VMPP dans le cadre de la consolidation dynamique et surpasse largement la qualité des plans de reconfiguration obtenus avec VMRP avec des coûts au moins 90% moindre. Ces gains nécessitent en contre-partie des temps de calcul plus long. Cependant, nos différentes expérimentations sur grappes ont montré que ces temps supplémentaires (de l'ordre de la minute) étaient compensés par la réduction du temps d'exécution des plans de reconfiguration (de l'ordre de 5 minutes). Finalement, notre évaluation du module de décision dédié à l'ordonnancement de tâches a également montré l'intérêt de notre approche pour le développement d'algorithmes d'ordonnancement efficaces basés sur une gestion dynamique des tâches.

Perspectives

De nouvelles expérimentations permettraient de compléter l'évaluation d'Entropy. Nous avons d'abord montré que la boucle d'auto-adaptation se devait d'être la plus rapide possible afin de supporter les besoins changeants des machines virtuelles. Il serait alors pertinent de réaliser de nouvelles expériences où l'on ferait varier la fréquence de changement de phases des applications afin d'observer avec précision les limites de la réactivité d'Entropy. Nous avons également observé que l'évaluation du module dédié à l'ordonnancement est plus délicate que l'évaluation du module dédié à la consolidation dynamique. L'évaluation actuelle ne montre pas les limites des possibilités de notre module et pour évaluer plus finement celui-ci, il serait pertinent de se servir de traces réelles d'exécution de tâches. Plusieurs sites proposent de telles traces [PWA, GWA] mais celles-ci ne sont pas réellement exploitables dans notre situation : sans connaissances de l'utilisation réelle des ressources des différentes tâches, nous devons nous borner à supposer une utilisation constante de celles-ci. Dans ce cas, l'intérêt d'une gestion dynamique des tâches dépend des spécificités de l'ordonnanceur. Si celui-ci exécute les tâches séquentiellement dans un ordre de type FCFS par exemple, alors nous ne pourrons utiliser que la migration de machines virtuelles pour améliorer la consolidation. Avec des traces plus complètes, indiquant la charge des tâches en fonction du temps, il serait possible de ré-exécuter ces traces, sur un simulateur ou sur une grappe, en comparant leur exécution avec différentes stratégies d'ordonnancement pour Entropy.

Finalement, une limitation actuelle de notre évaluation provient du décalage important dans la taille des problèmes traités dans les micro-évaluations et l'évaluation sur grappes. Dans les micro-évaluations des modules VMPP et VMRP, nous utilisons des configurations possédant jusqu'à 400 machines virtuelles et 200 nœuds, les expérimentations sur grappes utilisent des configurations contenant au plus 35 nœuds de calcul et autant de machines virtuelles. Les premières expérimentations sur grappes d'Entropy ont été réalisées en août 2008 sur la plateforme Grid'5000. À ce moment, le déploiement des hyperviseurs Xen sur les nœuds n'était pas supporté officiellement et seules 2 grappes le permettaient. Aujourd'hui, la totalité des sites supportent le déploiement de cet hyperviseur et des expériences à plus large échelle peuvent être réalisées afin d'harmoniser les différentes évaluations.

Si nous nous focalisons sur les capacités de passage à l'échelle d'Entropy et plus spécialement du problème VMRP, nous pensons que le calcul de plans de reconfiguration sur des grappes possédant plus de quelques centaines de nœuds nécessitera un temps de calcul trop long. Si nous souhaitons améliorer ce passage à l'échelle, il importe de revoir le modèle architectural décrivant les différents nœuds. Une solution consisterait à proposer un modèle de résolution hiérarchique possédant un grain variable. Un premier module se focaliserait sur le placement de tâches sur des ensembles de nœuds (représentant une armoire, une grappe ou un site par exemple). Puis pour chacun de ces ensembles, un sous-module se focaliserait sur l'affectation des machines virtuelles composant les tâches sur les nœuds composant l'ensemble. Si une tâche ne peut être hébergée entièrement sur un ensemble, alors celle-ci serait découpée pour être hébergée sur plusieurs ensembles si les contraintes de placement l'autorisent. Nous pensons que cette décomposition du problème de placement global en plusieurs sous-problèmes améliorerait le passage à l'échelle d'Entropy en dégradant faiblement ces performances. D'autant plus que ce changement de granularité offrirait gratuitement une distribution et une parallélisation du calcul des affectations dans les différents sous-ensembles.

Chapitre 11

Conclusion

11.1 Bilan

Nous avons étudié dans cette thèse les problèmes liés à la gestion dynamique des tâches dans les grappes. Cette approche permet à des stratégies d'ordonnancement de manipuler les tâches à la volée en utilisant des mécanismes de préemption et de migration des tâches et la réallocation de ressources. L'utilisation de tels mécanismes assure une meilleure utilisation des ressources dans les grappes. Nous avons cependant mis en avant que cette approche était peu utilisée en pratique. En effet, il est d'abord complexe d'implémenter d'une manière efficace ces mécanismes. L'approche actuelle qui consiste à exécuter chaque tâche directement sur des nœuds et à manipuler ses processus rend les opérations de préemption et de migrations délicates. La gestion des dépendances résiduelles ne permet pas de manipuler l'environnement d'exécution de la tâche tandis que le maintien des connexions réseau ou des descripteurs de fichier ouverts n'est pas évident. De plus, l'exécution d'une tâche peut nécessiter une adaptation plus ou moins lourde de celle-ci à l'environnement d'exécution de la grappe. Enfin, l'ordonnancement et le placement des tâches sont soumis à de nombreuses contraintes, spécifiques à chaque grappes de serveurs et à chaque tâche exécutées. Ces contraintes sont difficilement prisent en compte dans les stratégies actuelles.

Pour faciliter le développement de stratégies de gestion dynamique des tâches, nous avons d'abord proposé un support dédié à l'exécution des tâches. La virtualisation permet d'isoler fortement tout un système dans une machine virtuelle, dont les accès aux ressources sont contrôlés par un hyperviseur. L'utilisateur peut alors exécuter ses tâches dans leur environnement logiciel d'origine en déployant chacun de leurs composants dans une machine virtuelle. Un hyperviseur, installé sur chaque nœud de calcul, exécute les différentes machines virtuelles et supporte une manipulation de celles-ci compatible avec les besoins d'une gestion dynamique des tâches.

Pour satisfaire le besoin de flexibilité dans la définition et la résolution de stratégies d'ordonnancement et de placement, nous proposons une approche basé sur la Programmation Par Contraintes (PPC). Cette approche générique de résolution de problèmes combinatoires permet de modéliser naturellement un problème comme une conjonction de contraintes prédéfinies. Un solveur calcule alors une solution en tentant de satisfaire simultanément toutes les contraintes.

Pour valider notre approche, nous avons implémenté le prototype Entropy, un environnement autonome écrit en Java, dédié à la manipulation de tâches composées de machines virtuelles. Cet environnement dispose d'interfaces pour s'intégrer avec différents systèmes de supervision et hyperviseurs et propose une séparation entre un module de décision dédié au calcul de la configuration à obtenir pour avoir un ordonnancement optimal et un module de planification des actions assurant la transition vers cette nouvelle configuration depuis la configuration courante. Le module de décision peut être définis par les administrateurs de la grappe et adapté aux spécificités de celle-ci. Le module de planification est quant à lui plus générique et considère les problèmes de dépendances relatifs à l'exécution des actions de manipulation des machines virtuelles et leurs coûts en temps et en performance. Ce module calcule alors un plan de reconfiguration assurant la faisabilité de chacune des actions tout en exécutant celles-ci le plus tôt possible avec un degré de parallélisme maximum. Cette séparation entre le module de décision et de planification permet aux administrateurs de se focaliser uniquement sur les problèmes liés à

l'ordonnancement des tâches sans avoir à considérer les problèmes liés au processus de reconfiguration.

Nous avons évalué la faisabilité de notre approche en développant au dessus d'Entropy deux exemples pratiques de modules de décision. Le module dédié à la consolidation dynamique gère le placement d'un ensemble de machines virtuelles ayant des besoins en ressources CPU variables sur un nombre minimum de nœuds. Le second module de décision est dédié à l'ordonnancement et permet d'écrire facilement des stratégies d'ordonnancement sélectionnant les tâches à exécuter sur une grappe. La généricité de notre module de planification en aval, supporte ainsi l'exécution de stratégies à base de partitionnement spatial ou temporel, et considère que les tâches sont préemptibles.

Dans cette thèse, nous avons pu valider expérimentalement nos hypothèses en évaluant les différents composants d'Entropy ainsi que les deux cas d'utilisation implémentés. Une première série de micro-évaluations a étudié le comportement des modules de résolution d'Entropy selon différentes caractéristiques des configurations. Ces évaluations ont pu estimer les capacités de passage à l'échelle de notre approche ainsi que le temps moyen pour la résolution de ces problèmes d'optimisation complexes. L'évaluation sur grappe de notre stratégie de consolidation dynamique a démontré qu'elle réduisait efficacement le nombre de nœuds nécessaire à l'hébergement de machines virtuelles ayant des besoins en ressources variables. De la même manière, l'évaluation de notre module de décision dédié à l'ordonnancement de tâches a validé notre approche autant pour la facilité de développement d'un ordonnanceur complexe, que pour la qualité des solutions produites. Dans chacun des cas, ces évaluations ont montré l'efficacité de notre méthode pour réduire le temps moyen d'une reconfiguration, critère majeur pour une auto-adaptation rapide.

Nous avons donc estimé dans le cadre de cette thèse que notre approche à base de machines virtuelles et de programmation par contraintes était une solution viable pour faciliter et améliorer le développement de gestionnaires de ressources basés sur une gestion dynamique des tâches. D'une part nous avons montré que notre infrastructure supportait le développement de stratégies de placement ou d'ordonnancement complexes. D'autre part, nous avons évalué l'intérêt de minimiser le temps de reconfiguration d'un tel environnement afin d'assurer la plus grande réactivité possible.

11.2 Perspectives

Tout au long de cette thèse, nous avons discuté de certaines limitations et de perspectives concernant l'architecture générale et les différents composants d'Entropy. Nous reprenons dans cette section les perspectives que nous jugeons les plus pertinentes en les envisageant de manière plus globale et à plus long terme.

11.2.1 Consolidation dynamique

Notre module de consolidation dynamique assure une concentration optimale des machines virtuelles sur un nombre minimum de nœuds. Si cette approche répond à un besoin réel, on remarque cependant que son intégration dans des centres d'hébergement d'applications ou même sur des grappes de production est limitée. Dans le cadre de l'hébergement de services web par exemple, les applications distribuées hébergées ont généralement des contraintes relatives à la tolérance aux pannes. Traditionnellement, la solution consiste à introduire de la redondance dans le service, en hébergeant plusieurs instances d'un même service sur des nœuds différents. En amont, un ou plusieurs répartiteurs de charge distribuent les requêtes HTTP sur les différentes instances. De cette manière, si un nœud hébergeant une instance tombe en panne, alors le service reste fonctionnel car plusieurs instances sont encore disponibles. Dans le cadre de l'utilisation de la consolidation dynamique, l'algorithme de placement peut être amené à héberger toutes les instances d'un service sur un même nœud. Si ce nœud venait à tomber en panne alors le service entier serait interrompu. Nous pensons que la consolidation dynamique telle qu'elle est appliquée dans Entropy mais également dans la littérature actuelle réduit la tolérance aux pannes en ne considérant pas le besoin de séparer certains composants. Ces contraintes sont cependant primordiales dans les services à haute disponibilité actuels. L'approche flexible retenue pour le développement d'Entropy permet de considérer facilement ces contraintes liées à la tolérance aux pannes. Notre base de connaissances dispose déjà de la contrainte $assignOnDifferentNode$. Celle-ci assure que chaque machine virtuelle d'un ensemble donné sera hébergée sur un nœud différent (sa définition complète est disponible en annexe). Ainsi, lorsqu'un

utilisateur soumet une tâche contenant des services redondants, il déclare en complément des besoins initiaux en ressources, les différents ensembles de machines virtuelles concernées. Il est alors possible de spécialiser VMPP pour ajouter une contrainte $assignOnDifferentNode$ sur chacun des ensembles de machines virtuelles. Ce module de consolidation modifié assurera alors l'utilisation d'un nombre minimal de nœuds tout en maintenant les contraintes liées à la tolérance aux pannes.

11.2.2 Ordonnancement flexible

Si le développement de stratégies d'ordonnancement est simple et rapide pour une personne connaissant l'API d'Entropy, elle nécessite tout de même des compétences dans le langage de programmation JAVA et une connaissance du fonctionnement d'Entropy. Ces points peuvent freiner les administrateurs dans le développement d'ordonnanceurs et mener à des erreurs de conceptions liées au manque d'expertise dans l'API d'Entropy et à l'absence de vérifications des algorithmes d'ordonnancement. Ce problème est d'autant plus contraignant que certaines erreurs ne seront détectables par le développeur qu'après l'observation du comportement d'Entropy lors de son exécution. Une approche pour l'écriture d'ordonnanceur pour Entropy dans un langage dédié (DSL) pourrait être une solution à ces problèmes. Un DSL est un langage créée pour résoudre des problèmes spécifiques à un domaine. Le langage SQL [MS01] par exemple est un langage dédié à la manipulation de bases de données relationnelles. Les langages dédiés s'opposent à des langages généralistes comme JAVA ou C couvrant un large spectre de domaines. L'utilisation d'un DSL dédié à l'écriture de stratégies d'ordonnancement pour Entropy permettrait de réduire le niveau d'expertise nécessaire à leur écriture tout en permettant la vérification de certaines propriétés de corrections avant l'exécution de la stratégie. Le développeur manipule en effet uniquement des concepts liés à l'ordonnancement des tâches et à la génération d'une configuration avec un langage restreint. Le code de cette stratégie est ensuite compilé pour être incorporé dans Entropy uniquement si celui-ci respecte des propriétés telles que l'impossibilité de détruire involontairement des tâches par exemple.

Par ailleurs, l'approche actuelle d'ordonnancement se limite à la déclaration de la liste des tâches à exécuter. Il serait pertinent de pouvoir spécifier également des règles de placement entre les différentes machines virtuelles des tâches. On retrouve alors un cas d'utilisation similaire à la consolidation dynamique où l'utilisateur peut souhaiter exécuter chacune de ses machines virtuelles sur des nœuds différents. Mais il importe également de pouvoir exécuter plusieurs machines virtuelles sur un même nœud, lorsque celles-ci sont fortement inter-communicantes par exemple ou interdire d'exécuter certaines machines virtuelles sur une liste de nœuds. La base de connaissances d'Entropy dispose déjà d'un ensemble de contraintes pré-implémentées répondant à ces besoins (l'annexe regroupe et définit ces différentes contraintes). Notre DSL pour l'ordonnancement pourrait alors intégrer la possibilité d'exprimer ces contraintes de placement pour affiner l'ordonnancement et le placement des machines virtuelles.

11.2.3 Reconfiguration

Dans cette thèse, nous avons identifié différentes perspectives propres au module de planification de la reconfiguration, mais également liées aux besoins, non-considérés à l'origine, des modules de décisions. Notre algorithme de création de plans de reconfiguration nécessite qu'un nœud de la grappe dispose d'un espace libre suffisamment grand pour servir de pivot en cas de dépendances cyclique. Si cette situation est envisageable dans la majorité des cas, il est intéressant de proposer une alternative à base de suspension et de reprise de machine virtuelle. Lorsqu'un cycle de dépendances n'est pas résolvable, nous réalisons une suspension d'un machine virtuelle puis sa reprise lorsque les dépendances ont été résolues. Cette solution assure la faisabilité d'une reconfiguration dans toutes les situations, sans avoir recours à la préemption d'une machine virtuelle pour résoudre tous les problèmes de cycle de dépendances [GIYC06], solution à la fois plus coûteuse en temps et en performances.

Par ailleurs, notre étude des contextes des actions a mis en avant l'intérêt de considérer le temps d'exécution des actions, ainsi que leur impact sur les performances des nœuds impliqués. Nous avons implémenté une première fonction de coût minimisant le temps d'exécution d'un plan, mais il serait pertinent de réaliser également une fonction de coût dédiée à la réduction de l'impact d'une reconfiguration sur les performances de la grappe. La confrontation de ces deux approches pour l'optimisation de la reconfiguration permettrait de choisir l'approche la plus adaptée en fonction de différents paramètres tels que le type de tâches (du calcul scientifique par exemple) et la fréquence de variation des besoins en

ressources. Cette confrontation pourrait également mettre en avant un intérêt pour une fonction de coût réalisant par pondération un compromis entre la réduction du temps d'exécution d'un plan et son impact sur les performances.

Finalement, nous avons démontré dans cette thèse que la PPC était une solution efficace pour la résolution de problèmes d'ordonnancement variés. La planification des actions est assimilable à un problème d'ordonnancement, où nous ordonnons les actions à exécuter pour assurer leur faisabilité tout en proposant un coût global minimum. L'approche que nous avons choisi dans cette thèse est une approche mixte où le calcul d'un plan de reconfiguration depuis une configuration source vers une configuration finale est réalisé de manière heuristique tandis que l'algorithme d'optimisation VMRP repose sur la PPC. Le problème VMRP maintient l'état de toutes les machines virtuelles ainsi que la viabilité de la configuration finale tandis que l'heuristique de création de plan assure la viabilité de la configuration durant tout le processus de reconfiguration. Nous avons expliqué dans les perspectives liées à la consolidation dynamique et à l'ordonnancement des tâches que la prise en compte des contraintes de placement dans le module de décision est une contribution pertinente. Seulement, notre approche actuelle pour la reconfiguration n'assure pas le maintien de ces contraintes de placement durant le processus de reconfiguration. L'approche PPC pour la définition du problème VMRP permet de considérer ces contraintes facilement, cependant notre heuristique ne peut assurer que ces contraintes seront maintenues durant tout le processus de reconfiguration. Une solution consisterait alors à utiliser une approche basée uniquement sur de la PPC pour la planification et l'optimisation du processus de reconfiguration.

11.2.4 Architecture générale d'Entropy

Dans cette thèse, nous avons considéré uniquement les capacités et les besoins en ressources CPU et mémoire des nœuds et des machines virtuelles. Cette considération n'est pas suffisamment réaliste lorsque les applications font une utilisation conséquente du réseau. Il serait en effet utile de pouvoir spécifier des règles de placement relatives à des contraintes liées au réseau afin de rapprocher des machines virtuelles fortement communicantes par exemple ou de s'assurer que la latence entre deux machines virtuelles est suffisante pour exécuter des applications temps-réel. Il se pose alors le problème de la modélisation d'un réseau et de son utilisation dans Entropy. Nous devons d'abord considérer la capacité réseau de chaque nœud, de chaque lien et de chaque équipement d'interconnexion. Il est ensuite nécessaire d'identifier et de quantifier le trafic réseau entre chaque machine virtuelle. Passé cette étape, il convient alors de modéliser finement l'impact sur le trafic réseau de l'affectation d'une machine virtuelle sur un nœud. Cette phase nécessite de tenir compte de la topologie du réseau et des règles de routage. La modélisation peut être simplifiée dans le cadre d'un réseau uniforme en étoile avec des règles de routage statiques par exemple. Elle peut être cependant beaucoup plus délicate à modéliser si la topologie est complexe (un réseau cubique par exemple) ou si le routage est dynamique (s'il s'adapte à la charge courante des différents liens réseaux par exemple). L'implémentation d'un tel modèle pourrait être simplifiée par notre approche PPC car elle n'implique pas de modifier les différentes contraintes de la base de connaissances. Il suffirait uniquement de créer de nouvelles variables décrivant les ressources et des contraintes de distances et de graphes, dédiées à la mise à jour de ces variables en fonction de l'affectation des machines virtuelles.

L'informatique autonome propose entre autre une séparation de certains concepts durant une phase d'auto-adaptation : un module de décision choisit les adaptations à réaliser tandis qu'un module de planification prépare l'auto-adaptation. Le développement d'Entropy suit cette logique et propose ainsi plusieurs modules de décision et un module de planification. Cependant, en comparant par exemple VMPP et VMRP, on observe que les définitions de chaque problème sont quasiment équivalentes. La définition de VMPP est focalisée sur la réduction du nombre de nœuds utiles. La définition de VMRP se focalise quant à elle sur la réduction du coût du plan associé à la configuration résultat mais utilisant un nombre de nœuds égal à la solution de VMPP. On peut donc considérer que la résolution de VMPP ne sert qu'à préparer le problème VMRP. Cette remarque est également valable avec le problème RJAP qui ne sert qu'à indiquer à VMRP l'état des tâches pour la configuration finale. Nous avons donc à résoudre deux problèmes dont l'écriture est quasiment équivalente pour calculer une nouvelle configuration. En théorie, cette approche est coûteuse et réduit significativement la réactivité de l'auto-adaptation. Il serait possible de ne résoudre qu'un seul problème en considérant un problème multi-objectif. Résoudre un tel problème consiste à calculer un compromis entre plusieurs variables objectif (par exemple le nombre minimum de

nœuds pour héberger les machines virtuelles et le coût de reconfiguration) dont la valeur est pondérée. Des évaluations de précédentes implémentations d'Entropy ont cependant montré que le temps de résolution d'un tel problème ainsi que la qualité du résultat étaient moins satisfaisants que la résolution de deux problèmes indépendants. Améliorer l'efficacité de la résolution de problèmes d'optimisation multi-objectifs serait alors bénéfique à Entropy. Cette perspective couvre cependant un champ d'application beaucoup plus large que celui de notre thèse.

Entropy propose un environnement complet permettant de contrôler les machines virtuelles durant la totalité de leur cycle de vie. Cependant la contribution de cette thèse se focalise sur l'aspect décisionnel et la planification de l'auto-adaptation. La nécessité de maintenir la totalité d'un environnement de gestion des machines virtuelles est potentiellement un frein au déploiement de nos contributions dans des environnements de productions. Utiliser des machines virtuelles dans une grappe nécessite en effet de revoir l'architecture de celle-ci, notamment au niveau du stockage ou des plans d'adressage du réseau. Récemment, des suites logicielles comme OpenNebula [Mon08] proposent de faciliter la mise en place de tels environnements. Afin de favoriser l'utilisabilité d'Entropy, il serait important de choisir après une étude des différents environnements disponibles, un environnement de référence servant de support à notre contribution. Cette intégration dans un environnement ouvert et disposant d'une communauté de développeurs et d'utilisateurs provenant du monde académique et du monde industriel nous permettrait à la fois une meilleure compréhension des besoins des utilisateurs mais également une visibilité supérieure de notre contribution.

Chapitre 12

Annexe

DURANT cette thèse, nous avons développé différentes contraintes agissant sur l'état des tâches et le placement des machines virtuelles en cours d'exécution. Ces différentes contraintes sont incluses en standard dans la base de connaissances d'Entropy et peuvent être utilisées pour définir des problèmes spécifiques à une grappe. Cette annexe liste les différentes contraintes développées ainsi que leur définition à la fois textuelle et formelle en suivant le modèle VMAP et les contraintes globales du solveur Choco.

12.1 Contraintes liées à l'ordonnancement des machines virtuelles

Désignation	Définition
$mustBeReady(j)$	Indique qu'une machine virtuelle d'indice j doit être dans le pseudo-état `Prêt`. Le solveur choisira l'état concret en fonction de son état courant. Si celle-ci est dans l'état `Exécution` alors son état sera `Endormi`, sinon son état sera `Attente`. Expression : $v_j = n + 1$
$mustBeRunning(j)$	Indique qu'une machine virtuelle d'indice j doit être dans l'état `Exécution`. Expression : $v_j \leq n$ ou $v_j \neq n + 1 \wedge v_j \neq n + 2$
$mustBeStopped(j)$	Indique qu'une machine virtuelle d'indice j doit être dans l'état `Terminé`. Expression : $v_j = n + 2$

12.2 Contrainte liée à l'accessibilité aux ressources des machines virtuelles

Désignation	Définition
$setKnapsackOn(i, pRc, vRc)$	Établit une règle de sac à dos sur le nœud d'indice i établissant que la capacité du nœud pour la ressource pRc est supérieure à la somme des besoins des machines virtuelles hébergées pour la ressource vRc Expression (pour les ressources CPU) : $\mathcal{R}_c \cdot H_i \leq c_c^i$

12.3 Contraintes liées au placement des machines virtuelles

Désignation	Définition
$assignOnDifferentNode(j_0, \ldots, j_k)$	Indique que toute les machines virtuelles virtuelle dont l'indice j est passé en paramètre seront hébergées sur un nœud différent. Expression[1] : $allDifferent(v_{j_0}, \ldots, v_{j_k})$
$denyOnNode(i, j)$	Indique que la machine virtuelle d'indice j ne devra pas être hébergée sur le nœud d'indice i. Expression : $v_j \neq i$
$assignOnSameNode(j_0, \ldots, j_k)$	Indique que toute les machines virtuelles dont l'indice j est passé en paramètre seront hébergées sur un même nœud. Expression[1] : $allEquals(v_{j_0}, \ldots, v_{j_k})$
$setMaximumVMsOnNode(i, nb)$	Indique que le nœud d'indice i ne peut héberger simultanément au plus que nb machines virtuelles. Expression : $\sum_{i \in [0,k]} H_i \leq nb$
$atMostNValue(n, j_0, \ldots, j_k)$	Indique que le nombre de nœuds hébergeant toute les machines virtuelles dont l'indice j est passé en paramètre doit être inférieur à n. Expression[1] : $atmost\_nvalue(n, v_{j_0}, \ldots, v_{j_k})$

12.4 Contraintes liées à l'optimisation de la résolution de problèmes

Désignation	Définition
$SymetryBreaking$	Considère les nœuds et les machines virtuelles équivalentes pour réaliser un filtrage supplémentaire lorsqu'une affectation n'est pas consistante. Expression : $\forall a, b \in [1, n], \forall x, y \in [1, k], \mathrm{VM}_x \equiv \mathrm{VM}_y, \mathrm{N}_a \equiv \mathrm{N}_b,$ $a \notin \mathcal{D}(v_x) \Rightarrow a \notin \mathcal{D}(v_y) \wedge b \notin \mathcal{D}(v_x) \wedge b \notin \mathcal{D}(v_y)$

[1] Ces contraintes globales sont disponibles dans le solveur de contraintes Choco.

Bibliographie

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII : Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[AHH+08] Eric Anderson, Joseph Hall, Jason Hartline, Mick Hobbes, Anna R. Karlin, Jared Saia, and John Wilkes. Algorithms for data migration. *Algorithmica*, aug 2008.

[AMD05] Advanced micro devices. amd64 virtualization codenamed "pacifica" technology, secure virtual machine architecuture reference manual, may 2005.

[AMZ03] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. In *SPAA '03 : Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–265, New York, NY, USA, 2003. ACM.

[BCC+06] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lantéri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, November 2006.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet : A gigabit-per-second local area network. *IEEE Micro*, 15(1) :29–36, 1995.

[BCR05] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog. Technical Report T2005-08, Swedish Institute of Computer Science, 2005.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003. ACM Press.

[BDH03] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet : The google cluster architecture. *IEEE Micro*, 23(2) :22–28, 2003.

[BKB07] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing SLA violations. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, May 2007.

[BS93] G. C. Buttazzo and Jack A. Stankovic. Re : Robust earliest deadline scheduling. Technical report, Amherst, MA, USA, 1993.

[CDCG+05] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGRID '05 : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 776–783, Washington, DC, USA, 2005. IEEE Computer Society.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 273–286, Boston, MA, USA, May 2005.

[CGJ96] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing : a survey. *Approximation algorithms for NP-hard problems*, pages 46–93, July 1996.

[CIG+03] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *HPDC '03 : Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 90, Washington, DC, USA, 2003. IEEE Computer Society.

[CJLR06] Hadrien Cambazard, Narendra Jussien, François Laburthe, and Guillaume Rochart. The choco constraint solver. In *INFORMS Annual meeting*, Pittsburgh, PA, USA, November 2006.

[Con08] Repairing Bin Packing Constraints. Fukunaga, alex s. In *First Workshop on Bin Packing and Placement Constraints BPPC'08*, may 2008.

[Cre81] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5) :483–491, 1981.

[DGBL96] Xiaotie Deng, Nian Gu, Tim Brecht, and KaiCheng Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *SODA '96 : Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 159–167, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[EKR02] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Proceeding of the second Workshop on Power Aware Computing Systems*, pages 179–196, Cambridge, MA, USA, 2002.

[ENT] Homepage of entropy. `http://entropy.gforge.inria.fr`.

[ES06] Wesley Emeneker and Dan Stanzione. Increasing reliability through dynamic virtual clustering. In *High Availability and Performance Computing Workshop*, 2006.

[ET05] Yoav Etsion and Dan Tsafrir. A short survey of commercial cluster batch schedulers. Technical Report 2005-13, The Hebrew University of Jerusalem, may 2005.

[FdW02] Michael Frumkin and Rob F. Van der Wijngaart. NAS grid benchmarks : A tool for grid space exploration. *Cluster Computing*, 5(3) :247–255, 2002.

[Fei90] Dror G. Feitelson. Distributed hierarchical control for parallel processing. *Computer*, 23(5) :65–77, 1990.

[Fei96] Dror G. Feitelson. Packing schemes for gang scheduling. In *IPPS '96 : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110, London, UK, 1996. Springer-Verlag.

[FHN+03] Wu-chun Feng, Justin (Gus) Hurwitz, Harvey Newman, Sylvain Ravot, R. Les Cottrell, Olivier Martin, Fabrizio Coccetti, Cheng Jin, Xiaoliang (David) Wei, and Steven Low. Optimizing 10-gigabit ethernet for networks of workstations, clusters, and grids : A case study. In *SC '03 : Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 50, Washington, DC, USA, 2003. IEEE Computer Society.

[FJ97] Dror G. Feitelson and Morris A. Jette. Improved utilization and responsiveness with gang scheduling. In *IPPS '97 : Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 238–261, London, UK, 1997. Springer-Verlag.

[FJ99] Howard Frazier and Howard Johnson. Gigabit ethernet : From 100 to 1,000 mbps. *IEEE Internet Computing*, 3(1) :24–31, 1999.

[FPSF06] Niels Fallenbeck, Hans-Joachim Picht, Matthew Smith, and Bernd Freisleben. Xen and the art of cluster scheduling. In *VTDC '06 : Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 4, Washington, DC, USA, 2006. IEEE Computer Society.

[FR96] Dror G. Feitelson and Larry Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *IPPS '96 : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, London, UK, 1996. Springer-Verlag.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *IPPS '97 : Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–34, London, UK, 1997. Springer-Verlag.

[GEC06] Gecode : Generic constraint development environment. `http://www.gecode.org`, 2006.

[Gen01] W. Gentzsch. Sun grid engine : Towards creating a compute power grid. In *CCGRID '01 : Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[GIMS07] Laura Grit, David Irwin, Varun Marupadi, and Piyush Shivam. Harnessing virtual machine resource control for job management. In *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2007.

[GIYC06] Laura Grit, David Irwin, Aydan Yumerefendi, and Jeff Chase. Virtual machine hosting for networked clusters : Building the foundations for "autonomic" orchestration. In *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, pages 1–8, November 2006.

[GLV+08] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine : Harnessing memory redundancy in virtual machines. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI 2008)*. USENIX, December 2008.

[GWA] The grid workload archive.

[Hab08] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166) :8, 2008.

[HD06a] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *in Proceedings of SciDAC. 2006*, page 2006. Online]. Available : http ://s-tacks.iop.org/17426596/46/494, 2006.

[HD06b] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics : Conference Series*, 46 :494–499, 2006.

[HE80] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3) :263–313, October 1980.

[HG09] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, New York, NY, USA, 2009. ACM.

[HGM+07] Eric Harney, Sebastien Goasguen, Jim Martin, Mike Murphy, and Mike Westall. The efficacy of live virtual machine migrations over the internet. In *VTDC '07 : Proceedings of the 3rd international workshop on Virtualization technology in distributed computing*, pages 1–7, New York, NY, USA, 2007. ACM.

[HHK+01] Joseph Hall, Jason Hartline, Anna R. Karlin, Jared Saia, and John Wilkes. On algorithms for efficient data migration. In *SODA '01 : Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 620–629, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[hKW00] Poul henning Kamp and Robert N. M. Watson. Jails : Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[HR85] Frederick Hayes-Roth. Rule-based systems. *Commun. ACM*, 28(9) :921–932, 1985.

[ICG+06] David Irwin, Jeffrey Chase, Laura Grit, Aydan Yumerefendi, David Becker, and Kenneth G. Yocum. Sharing networked resources with brokered leases. In *ATEC '06 : Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[ID07] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on nagios. In *GMW '07 : Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28, New York, NY, USA, 2007. ACM.

[ILO] Ilog solver 5.0 : Reference manual. gentilly, france. `http://www.ilog.com`.

[JSC01] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the maui scheduler. In *JSSPP '01 : Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK, 2001. Springer-Verlag.

[KBKK06] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381, 2006.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[KS02] Michael Kozuch and M. Satyanarayanan. Internet suspend/resume. In *WMCSA '02 : Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 40, Washington, DC, USA, 2002. IEEE Computer Society.

[LGP07] Gnu lesser general public license 3.0. `http://www.gnu.org/licenses/lgpl-3.0.txt`, jun 2007.

[Lif95] David A. Lifka. The anl/ibm sp scheduling system. In *IPPS '95 : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[LVT] libvirt : The virtualization api. `http://libvirt.org`.

[LxC] Linux containers. `http://lxc.sourceforge.net/`.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MAC+08] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax : virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4) :41–54, 2008.

[MCC04] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system : design, implementation, and experience. *Parallel Computing*, 30(7) :817 – 840, 2004.

[MDP+00] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3) :241–299, 2000.

[MGVV07] Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker. Usher : an extensible framework for managing custers of virtual machines. In *LISA'07 : Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–15, Berkeley, CA, USA, 2007. USENIX Association.

[Mon08] Rubén S. Montero. Opennebula : The open source virtual machine manager for cluster computing. In *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid*, Shanghai, China, 2008.

[MS01] Jim Melton and Alan Simon. *SQL :1999 : understanding relational language components.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[MSS08] Erwan Mellor, Richard Sharp, and David Scott. Xen management api v1.0.6. `http://wiki.xensource.com/xenwiki/XenApi?action=AttachFile&do=get&target=xenapi-1.0.6.pdf`, jul 2008.

[MT90] Silvano Martello and Paolo Toth. *Knapsack problems.* Wiley, 1990.

[MVZ93] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems (ToCS)*, 11(2) :146–178, 1993.

[nim] Nimbus. `http://workspace.globus.org/`.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *ATEC '05 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[NS07] Ripal Nathuji and Karsten Schwan. VirtualPower : Coordinated power management in virtualizaed entreprise systems. In *21st Symposium on Operating Systems Principles (SOSP)*, October 2007.

[NS08] Ripal Nathuji and Karsten Schwan. VPM tokens : virtual machine-aware power budgeting in datacenters. In *HPDC '08 : Proceedings of the 17th international symposium on High performance distributed computing*, pages 119–128, New York, NY, USA, 2008. ACM.

[NSL+06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel® virtualization technology : Hardware support for efficient processor virtualization. In *Intel Technology Journal*, volume 10, aug 2006.

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization.* Wiley-Interscience, New York, NY, USA, 1988.

[NWG+08] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid*, Shanghai, China, 2008.

[OVF09] Open virtualization format specification. `http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf`, feb 2009.

[OVZ] Openvz. `http://wiki.openvz.org/`.

[PBCH02] Edouardo Pinheiro, Richardo Bianchini, Enrique Carrera, and Taliver Heath. Dynamic cluster reconfiguration for power and performance. In L. Benini, M. Kandemir, and J. Ramanujam, editors, *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, 2002.

[PFH+02] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The quadrics network : High-performance clustering technology. *IEEE Micro*, 22(1) :46–57, 2002.

[Pfi01] Gregory F. Pfister. Aspects of the infiniband(tm) architecture. In *CLUSTER '01 : Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 369, Washington, DC, USA, 2001. IEEE Computer Society.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7) :412–421, 1974.

[PT04] Daniel Price and Andrew Tucker. Solaris zones : Operating system support for consolidating commercial workloads. In *LISA '04 : Proceedings of the 18th USENIX conference on System administration*, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association.

[PWA] The parallel workloads archive. http ://www.cs.huji.ac.il/labs/parallel/workload/.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *SSYM'00 : Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[RJXG05] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38(5) :63–69, May 2005.

[RRX+06] P. Ruth, Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 5–14, 2006.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[SBS+95] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf : A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[SCP+02] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI) :377–390, 2002.

[SCZL96] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The easy - loadleveler api project. In *IPPS '96 : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47, London, UK, 1996. Springer-Verlag.

[SGS96] Jaspal Subhlok, Thomas Gross, and Takashi Suzuoka. Impact of job mix on optimizations for space sharing schedulers. In *Supercomputing '96 : Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 54, Washington, DC, USA, 1996. IEEE Computer Society.

[Sha04] Paul Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 648–662. Springer, 2004.

[SKF08] Borja Sotomayor, Kate Keahey, and Ian Foster. Combining batch execution and leasing using virtual machines. In *HPDC '08 : Proceedings of the 17th international symposium on High performance distributed computing*, pages 87–96, New York, NY, USA, 2008. ACM.

[SMLF08] Borja Sotomayor, Ruben Montero Montero, Ignacio Martin Llorente, and Ian Foster. Capacity leasing in cloud systems using the opennebula engine. In *Cloud Computing and Applications 2008 (CCA08)*, 2008.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization : a scalable, high-performance alternative to hypervisors. In *EuroSys '07 : Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 275–287, New York, NY, USA, 2007. ACM.

[SS72] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3) :157–170, 1972.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9) :1278–1308, 1975.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[TDG+06] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees de Laat, Joe Mambretti, Inder Monga, Bas van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Gener. Comput. Syst.*, 22(8) :901–907, 2006.

[TOP] Top500 supercomputing. `http://top500.org`.

[Tri01] M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. In *Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-01)*, pages 113–124, 2001.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice : the condor experience : Research articles. *Concurrency and Computation : Practice and Experience*, 17(2-4) :323–356, 2005.

[UCL04] Gladys Utrera, Julita Corbalan, and Jesus Labarta. Implementing malleability on mpi jobs. In *PACT '04 : Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 215–224, Washington, DC, USA, 2004. IEEE Computer Society.

[VAN08a] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper : power and migration cost aware application placement in virtualized systems. In *Middleware '08 : Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[VAN08b] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of hpc applications. In *ICS '08 : Proceedings of the 22nd annual international conference on Supercomputing*, pages 175–184, New York, NY, USA, 2008. ACM.

[VBO] Virtualbox. `http://www.virtualbox.org/`.

[VMF07] Vmware virtual machine file system : Technical overview and best practices. `http://www.vmware.com/pdf/vmfs-best-practices-wp.pdf`, Jul 2007.

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI) :181–194, 2002.

[Wei98] A. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *IPPS '98 : Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 542, Washington, DC, USA, 1998. IEEE Computer Society.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI) :195–209, 2002.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service : a distributed resource performance forecasting service for metacomputing. *Future Generation Computer System*, 15(5-6) :757–768, 1999.

[WSVY07] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, 2007.

[WTLS+09] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies : exploiting page sharing for smart colocation in virtualized data centers. In *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 31–40, New York, NY, USA, 2009. ACM.

Glossaire

API *Application Programming Interface*. Une interface de programmation est un ensemble de fonctions, procédures ou classes mis à disposition des programmes informatiques par une bibliothèque logicielle, un système d'exploitation ou un service, 51

ARP *Address Resolution Protocol*. Protocole de résolution d'adresse permettant la traduction d'une adresse de la couche réseau (une adresse IP par exemple) en une adresse propre à la couche liaison de donnée (une adresse MAC par exemple), 25

Batch Scheduling Traitement de tâches par lots. Le gestionnaire de ressources dispose d'une liste de tâches à exécuter. Le traitement se fait automatiquement, l'ordre d'exécution peut éventuellement être strict et mélanger exécution séquentielle et parallèle, 14

compilateur JIT Technique visant à améliorer les performances des machines virtuelles applicatives en traduisant du pseudo code machine en code natif au moment de son exécution, 20

CPU *Central Processing Unit*. L'unité centrale de traitement est le composant central. Il interprète un ensemble d'instructions (ISA) et traite les données d'un programme, 43

FCFS *First Come, First Serve* Stratégie d'ordonnancement de tâches. Les tâches sont traitées selon leur date de soumission. Les tâches les plus anciennes sont exécutées en priorité, 14

HTTP *HyperText Transport Protocol*. Protocole de communication client/serveur développé pour le *web*. Son principe consiste à naviguer entre différentes ressources misent à disposition par des serveurs par le biais de liens hypertextes. Les navigateurs *web* sont des clients HTTP, 49

hypercall Appel utilisé dans les hyperviseurs réalisant de la para-virtualisation. Contrairement à un appel système qui est intercepté par le noyau, l'*hypercall* traverse le noyau du système d'exploitation de la machine virtuelle pour être traité directement par l'hyperviseur, 22

IP *Internet Protocol*. Protocole de niveau 3 du modèle OSI (couche réseau) et du modèle TCP/IP permettant un service d'adressage unique pour l'ensemble des terminaux connectés, 25

ISA *Instruction Set Architecture.* Ensemble de toutes les instructions qu'un processeur peut exécuter, 20

Multicast Méthode de diffusion permettant d'envoyer des paquets IP à un groupe de destinataires, préalablement abonnés à ce groupe. Ce protocole s'oppose au mode de diffusion par défaut *unicast* permettant l'envoie de paquets à un unique destinataire, 48

NFS *Network File System*. Protocole développé par Sun Microsystems qui permet à un ordinateur d'accéder à des fichiers via un réseau par le biais d'appels à des procédures distantes. Ce système de fichiers en réseau permet de partager des données principalement entre systèmes UNIX, 68

noyau Partie centrale des systèmes d'exploitation. Il gère les ressources de l'ordinateur et permet aux différents composants matériels et logiciels de communiquer entre eux, 20

PPC La Programmation Par Contraintes est une approche modélisant un problème sous la forme d'un ensemble de variables possédant des valeurs dans des domaines. Des contraintes restreignent ensuite les valeurs que peuvent prendre simultanément une ou plusieurs variables. Résoudre un tel problème consiste à calculer les différentes variables pour chaque variable qui satisfont simultanément toutes les contraintes, 27

RJAP *Running Job Assignment Problem*. Problème du calcule d'une configuration viable assurant qu'un ensemble de tâches fini sera dans l'état `Exécution` , 78

RRD *Round Robin Database.* Base de données permettant la sauvegarde et le tracé graphique de données chronologique. La base de données occupe un espace disque fixé lors de son initialisation en regroupant dynamiquement dans des intervalles temporelles plus grand les données les plus anciennes, 49

SAN *Storage Area Network*. Réseau spécialisé mutualisant des unités de stockage. L'accès aux à l'espace disque est réalisé directement en mode bloc par les différents serveurs qui peuvent alors utiliser leur propre système de fichiers, 43

SSH *Secured SHell*. Protocole de connexion sur une machine permettant d'exécuter des commandes à distance. Contrairement aux anciens programmes tels que *telnet* ou *rsh*, l'intégralité de la communication est chiffrée, 49

TCP *Transmission Control Protocol*, protocole de transport de paquets réseaux en mode connecté. Ce protocole supporte le déséquencement et la perte de paquets et permet de traiter une connexion réseau comme un flux d'octets., 79

TCP *Transport Control Protocol.* Protocole de transport de paquets réseau fiable en mode connecté assurant leur acquittement et maintenant leur séquencement, 48

VMPP *Virtual Machine Packing Problem*. Problème combinatoire lié au calcul d'une configuration viable utilisant un nombre de nœuds minimum pour héberger un ensemble fini de machines virtuelles, 57

VMRP *Virtual Machine Replacement Problem*. Problème de calcul d'une configuration viable dont le coût du plan de reconfiguration associé est le plus faible, 73

XML *eXtensible Markup Langage*. Langage de balisage extensible permettant de stocker ou de transférer des données structurées sous la forme d'une arborescence. Le langage est extensible dans le sens où il permet à la personne de définir ses propres balises et son espace de nommage par le biais d'un schéma, 49

XML-RPC Protocole client/serveur pour l'appel de procédures (*Remote procedure call*) à travers un réseau. Ce protocole, basé sur des flux XML, améliore l'interopérabilité entre les clients et les serveurs en supprimant toute dépendances avec un langage de programmation unique, 49

Gestion dynamique des tâches dans les grappes, une approche à base de machines virtuelles

Fabien Hermenier

Mots-clés : Grappes, Machines virtuelles, Gestion des ressources, Programmation par contraintes

Les gestionnaires de ressources reposant sur une gestion dynamique des tâches permettent une utilisation efficace des ressources des grappes de serveurs. Ils mettent en œuvre pour cela des mécanismes manipulant à la volée l'état des tâches et leur placement sur les différents nœuds de la grappe. En pratique, ces stratégies d'ordonnancement ad-hoc s'adaptent difficilement aux grappes. En effet, celles-ci ne permettent pas nécessairement une manipulation fiable des tâches et peuvent imposer des contraintes d'ordonnancement spécifiques.

Dans cette thèse, nous nous sommes fixés comme objectif de faciliter le développement de gestionnaires de ressources basés sur une gestion dynamique des tâches. Pour cela, nous avons retenu une architecture à base de machines virtuelles qui exécutent les tâches des utilisateurs dans leur propre environnement logiciel tout en proposant les primitives nécessaires à la manipulation de celles-ci de manière non-intrusive. Nous avons également proposé une approche autonome optimisant en continu l'ordonnancement des tâches. Les stratégies d'ordonnancement sont implémentées au moyen de la programmation par contraintes qui permet de définir de manière flexible des problèmes d'ordonnancement et de les résoudre.

Nous avons validé notre approche par le développement et l'évaluation du prototype Entropy, support pour l'implémentation de différentes stratégies d'ordonnancement. Celles-ci ont pu répondre efficacement à des problèmes concrets et actuels.

Online Management of Jobs in Clusters using Virtual Machines

Fabien Hermenier

Keywords : Clusters, Virtual Machines, Resources Management, Constraints Programming

Resources Management Systems relying on a dynamic management of jobs can effciently use resources in clusters. Indeed they provide mechanisms to manipulate online the state of the jobs and their assignment on the nodes. In practice, these scheduling strategies are hard to deploy on clusters as they can not necessarily handle the manipulation of the jobs and may have specific scheduling constraints to consider.

In this thesis, we try to ease the development of resources management systems relying on a dynamic management of jobs. We based our environment on the use of virtual machines to execute the jobs in their legacy environments while providing the mechanisms to manipulate them in a non-intrusive way. Moreover, we propose an autonomous environment to continuously optimize the scheduling of jobs. Scheduling strategies are implemented using constraints programming which aims to model and solve combinatory problems.

We validate our approach with the development of our prototype Entropy, which has been used to implement various scheduling strategies. The evaluation of these strategies show their capability to to solve present problems.